.Net : Entity Framework

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Plan

Plan

Entity Framework

Entity Framework

- **ORM** (**O**bject **R**elational **M**apping) officiel de **.NET Framework**.
- Développé par **Microsoft** en utilisant le langage **C#**.
- Permettant de manipuler des objets **C#** et sans écrire de requêtes **SQL**.
- Documentation officielle : `https://www.entityframeworktutorial.net/`

Entity Framework

Entity Framework

- **ORM** (**O**bject **R**elational **M**apping) officiel de **.NET Framework**.
- Développé par **Microsoft** en utilisant le langage **C#**.
- Permettant de manipuler des objets **C#** et sans écrire de requêtes **SQL**.
- Documentation officielle : `https://www.entityframeworktutorial.net/`

Autres **ORM** pour **C#**

- **NHibernate**
- **PetaPOCO**
- Ces deux frameworks sont inspirés par certains concepts **Java** comme le JavaBean

Entity Framework

ORM

- Programme informatique jouant le rôle du traducteur entre le modèle relationnel et le modèle objet.
- Objectif : plus de requêtes **SQL** dans les classes.
- Deux composants dans les **ORM** :
  - Entités (des classes à implémenter par le développeur) : qui représentent certaines tables.
  - Gestionnaire d'entités (une classe qui existe déjà) : à utiliser pour persister les entités dans la base de données.

Entity Framework

Dans le cas d'**Entity Framework**

- Entité = **POCO** + [classe de configuration ou décorateurs]
- Le gestionnaire d'entités : **Linq to Entities**

Entity Framework

Dans le cas d'**Entity Framework**

- Entité = **POCO** + [classe de configuration ou décorateurs]
- Le gestionnaire d'entités : **Linq to Entities**

POCO = **P**lain **O**ld **C**LR **O**bject

Entity Framework

Étapes à suivre

- Création d'un projet
- Intégration d'**Entity Framework** dans le projet
- Préparer le contexte
- Créer les entités
- Générer la base de données
- Manipuler les données avec **LINQ to Entities**

Entity Framework

Nouveau projet sous **Visual Studio Community 2022**

- Création d'une nouvelle solution (`CoursEntityFramework`).
- Création d'un projet Console (`CoursCodeFirst`)

Entity Framework

Pour intégrer `Entity Framework` dans le projet, on utilise `NuGet`.

Entity Framework

Pour intégrer `Entity Framework` dans le projet, on utilise `NuGet`.

`NuGet`

- Un gestionnaire de paquets, par défaut, pour **.NET**
- Open-source et gratuit
- Inclus dans *Visual Studio* depuis 2012
- Utilisable aussi en ligne de commande

Entity Framework

De quel paquets a t-on besoin ?

- **Microsoft.EntityframeworkCore.SqlServer** qui inclut déjà **Microsoft.EntityFrameworkCore**
- **Microsoft.EntityFrameworkCore.Tools** pour gérer les migrations

Entity Framework

Un paquet par SGBD

- Pour **SQLite** : **Microsoft.EntityFrameworkCore.Sqlite**
- Pour **MySQL** : **MySql.EntityFrameworkCore**
- Pour **PostgreSQL** : **Npgsql.EntityFrameworkCore.PostgreSQL**
- **...**

Entity Framework

Utiliser `NuGet` pour Télécharger les dépendances

- Faire clic droit sur `Dépendances` dans l'`Explorateur de solution`
- Choisir `Gérer les packages NuGet`
- Aller dans l'onglet `Parcourir` et chercher **Microsoft.EntityFrameworkCore.SqlServer** et **Microsoft.EntityFrameworkCore.Tools**
- Choisir la dernière version stable et installer
- Accepter, attendre la fin de l'installation
- Après installation, des packages relatifs à **Entity Framework** seront ajoutés dans `Dépendances/Packages` (Vérifier leur présence)

Entity Framework

On peut aussi installer les packages depuis un terminal

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Entity Framework

On peut aussi installer les packages depuis un terminal

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Et

```
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

Entity Framework

Créons une classe `ApplicationContext` qui hérite de `DbContext`

```
using Microsoft.EntityFrameworkCore;

namespace CoursCodeFirst
{
    internal class ApplicationContext : DbContext
    {
    }
}
```

Entity Framework

Créons une classe `ApplicationContext` qui hérite de `DbContext`

```
using Microsoft.EntityFrameworkCore;

namespace CoursCodeFirst
{
    internal class ApplicationContext : DbContext
    {
    }
}
```


Dans cette classe, on configure la connexion à la base de données et on déclare les entités que l'on souhaite utiliser

Entity Framework

Établissons la connexion avec la base de données

```
namespace TestEF
{
    internal class ApplicationContext: DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=FormationDb;
              Trusted_Connection=True;");
        }
    }

}
```

Entity Framework

Établissons la connexion avec la base de données

```
namespace TestEF
{
    internal class ApplicationContext: DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=FormationDb;
              Trusted_Connection=True;");
        }
    }

}
```


Explication

- `(localdb)` : une instance de **SQL Server** locale.
- `mssqllocaldb` : le nom de l'instance spécifique. Vous pouvez remplacer cela par l'adresse **IP** ou le nom du serveur **SQL Server** distant.
- `Database=NomDeLaBaseDeDonnees` : Spécifie le nom de la base de données à laquelle l'application se connecte.
- `Trusted_Connection=True` : Utilise l'authentification **Windows** pour se connecter à la base de données.

Entity Framework

Autres options de la chaîne de connexion

- `UserId` : nom de l'utilisateur de la base de données.
- `Password` : mot de passe de l'utilisateur de la base de données.

Entity Framework

Entité

- Classe **C#** déclarée dans `DbContext`.
- POCO : Version **.NET** d'un POJO **Java** (Plain Old Java Object).
- Pouvant contenir des données sur le mapping objet/relationnel.

Entity Framework

Créons une première entité `Personne` dans un répertoire `Models`

```
namespace CoursCodeFirst.Models
{
    public class Personne
    {
        public int Id { get; set; }
        public string Nom { get; set; }
        public string Prenom { get; set; }
        public int Age { get; set; }
    }
}
```

Entity Framework

Mettons à jour la classe `ApplicationContext.cs` **en déclarant un** `DbSet` **de l'entité** `Personne`

```
namespace TestEF
{
    internal class ApplicationContext: DbContext
    {

        public DbSet<Personne> Personnes { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=FormationDb;
              Trusted_Connection=True;");
        }

    }

}
```

Entity Framework

Quelques conventions

- Une entité déclarée dans `DbContext` aura une correspondance dans le modèle relationnel.
- Par défaut le nom de la table correspondante = `NomEntité + 's'`.
- Par défaut, **Entity Framework** considère l'attribut `Id` ou `NomEntite + Id` comme clé.
- Par défaut, **Entity Framework** crée les colonnes dans le même ordre que les attributs.

Entity Framework

Question

Si on voulait en dire plus ? sur la clé primaire, un nom différent pour la table et/ou les colonnes...

Entity Framework

Question

Si on voulait en dire plus ? sur la clé primaire, un nom différent pour la table et/ou les colonnes...

Deux solutions

- **Data Annotations** avec les décorateurs.
- **Fluent API**.

Entity Framework

Attentions aux types

- **Entity Framework** ne supporte pas certains types **C#** tels que `char`, `Object`...
- La liste des types compatibles :
  `https://www.devart.com/dotconnect/db2/docs/DataTypeMapping.html`

Entity Framework

Remarques

- Pour créer la base de données, il faut réaliser une migration.
- Pour réaliser une migration, on peut utiliser `Console du gestionnaire de paquet`.

Entity Framework

Remarques

- Pour créer la base de données, il faut réaliser une migration.
- Pour réaliser une migration, on peut utiliser `Console du gestionnaire de paquet`.

`Console du gestionnaire de paquet`

Dans le menu `Outils`, aller dans `Gestionnaire de package NuGet` et sélectionner `Console du gestionnaire de paquet`.

Entity Framework

Pour ajouter une migration, exécutez

```
Add-Migration InitialCreate
```

Entity Framework

Pour ajouter une migration, exécutez

```
Add-Migration InitialCreate
```

Constat

Un dossier `Migrations` a été créé contenant quelques fichiers.

Entity Framework

Pour ajouter une migration, exécutez

```
Add-Migration InitialCreate
```

Constat

Un dossier `Migrations` a été créé contenant quelques fichiers.

Pour exécuter la migration (et créer la base et la table), exécutez

```
Update-Database
```

Entity Framework

Dans `Main`, on utilise `Linq To Entities` pour persister les objets

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Nom = "Pradel",
        Prenom = "Jacques",
        Age = 45
    };

    // on persiste la personne
    context.Personnes.Add(personne);
    context.SaveChanges();
}
```

Entity Framework

Ou

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Nom = "Pradel",
        Prenom = "Jacques",
        Age = 45
    };

    // on persiste la personne
    context.Add(personne);
    context.SaveChanges();
}
```

Entity Framework

Remarques

- L'utilisation de `using` permet d'automatiser plusieurs opérations telle que la fermeture de la connexion...
- Le premier lancement du projet peut durer quelques dizaines de minutes : le temps de création de la base de données, la table...

Entity Framework

Pour vérifier que la base de données a bien été créée

- Aller dans le menu `Affichage` et cliquer sur `Explorateur d'objets SQL Server`
- Étendre la rubrique `SQL Server`
- Chercher la base de données (`CoursCodeFirst`) et vérifier la présence de la table `Personne`

Entity Framework

Pour insérer plusieurs personnes

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Nom = "Pradel",
        Prenom = "Jacques",
        Age = 45
    };

    var personne2 = new Personne()
    {
        Nom = "Scofield",
        Prenom = "Mickael",
        Age = 38
    };

    context.Personnes.AddRange(personne, personne2);
    context.SaveChanges();
}
```

Entity Framework

Dans le `Main`, on utilise `Linq To Entities` pour persister les objets

```
using (var context = new ApplicationContext())
{
    var personnes = context.Personnes.ToList();
    foreach(var elt in personnes)
    {
        Console.WriteLine($"Bonjour { elt.Prenom } { elt.Nom }");
    }
}
```

Entity Framework

Pour chercher une personne selon l'identifiant, on peut utiliser `Find`. Si aucun élément ne correspond, la valeur `null` sera retournée.

```
var personne2 = context.Personnes.Find(1);
Console.WriteLine(personne2.Nom);
// affiche Pradel
```

Entity Framework

Pour chercher une personne selon son nom

```
using (var context = new ApplicationContext())
{
    var personnes = context.Personnes
        .Where(p => p.Nom.Equals("Pradel"))
        .ToList();

    foreach (var elt in personnes)
    {
        Console.WriteLine($"Bonjour {elt.Prenom} {elt.Nom}");
    }
}
```

Entity Framework

Pour chercher une personne selon son nom (Le paramètre de la méthode est de type `FormattableString`, il doit donc commencer par `$`)

```
using (var context = new ApplicationContext())
{
    var personnes = context.Personnes
        .FromSql($"Select * from Personnes where prenom = 'Jean'")
        .ToList();

    foreach (var elt in personnes)
    {
        Console.WriteLine($"Bonjour {elt.Prenom} {elt.Nom}");
    }
}
```

Entity Framework

Pour éviter les injections SQL

```
using (var context = new ApplicationContext())
{
    string prenom = "Jean";
    var personnes = context.Personnes
        .FromSqlRaw($"SELECT * FROM Personnes WHERE Prenom = @prenom", prenom)
        .ToList();

    foreach (var elt in personnes)
    {
        Console.WriteLine($"Bonjour {elt.Prenom} {elt.Nom}");
    }
}
```

Entity Framework

Exercice

Écrire un code **Linq To Entities** qui permet de modifier une personne.

Entity Framework

Exercice

Écrire un code **Linq To Entities** qui permet de modifier une personne.

Exercice

Écrire un code **Linq To Entities** qui permet de supprimer une personne.

Entity Framework

Modification : correction

```
using (var context = new ApplicationContext())
{
    var p = context.Personnes.Find(1);
    Console.WriteLine($"{p.Id} {p.Nom} {p.Prenom}");

    p.Nom = "Drew";
    p.Prenom = "Nancy";

    context.SaveChanges();

    foreach (var elt in context.Personnes)
    {
        Console.WriteLine($"Bonjour {elt.Prenom} {elt.Nom}");
    }
}
```

Entity Framework

Suppression : correction

```
using (var context = new ApplicationContext())
{
    var p = context.Personnes.Find(2);
    context.Personnes.Remove(p);

    context.SaveChanges();

    foreach (var elt in context.Personnes)
    {
        Console.WriteLine($"Bonjour {elt.Prenom} {elt.Nom}");
    }
}
```

Entity Framework

Exemple avec `Detached` **et** `Added`

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Nom = "Pradel",
        Prenom = "Jacques",
        Age = 45
    };

    var objectEntry = context.Entry(personne);
    Console.WriteLine(objectEntry.State);
    // Detached

    context.Personnes.Add(personne);
    Console.WriteLine(objectEntry.State);
    // Added

    context.SaveChanges();
}
```

Entity Framework

Ou

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Nom = "Pradel",
        Prenom = "Jacques",
        Age = 45
    };

    var objectEntry = context.Entry(personne);
    Console.WriteLine(objectEntry.State);
    // Detached

    context.Personnes.Add(personne);
    context.ChangeTracker.DetectChanges();
    Console.WriteLine(context.ChangeTracker.DebugView.ShortView);
    // Added

    context.SaveChanges();
}
```

Entity Framework

Exemple avec `Deleted`

```
using (var context = new ApplicationContext())
{
    var p = context.Personnes.Find(7);
    context.Personnes.Remove(p);

    context.ChangeTracker.DetectChanges();
    Console.WriteLine(context.ChangeTracker.DebugView.ShortView);
    // Deleted

    context.SaveChanges();

}
```

Entity Framework

Exemple avec `Unchanged` **et** `Modified`

```
using (var context = new ApplicationContext())
{
    var p = context.Personnes.Find(8);
    var objectEntry = context.Entry(p);

    Console.WriteLine(objectEntry.State);
    // Unchanged

    p.Nom = "Drew";
    p.Prenom = "Nancy";

    context.ChangeTracker.DetectChanges();
    Console.WriteLine(context.ChangeTracker.DebugView.ShortView);
    // Modified

    context.SaveChanges();
}
```

Entity Framework

Pour modifier une entité sans la récupérer de la base de données, on peut l'attacher au contexte en spécifiant l'état `Modified`

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Id = 5,
        Nom = "Risch",
        Prenom = "Cyril",
        Age = 45
    };
    context.Entry(personne).State = EntityState.Modified;


    context.SaveChanges();
    foreach (var elt in context.Personnes)
    {
       Console.WriteLine($"{elt.Id} {elt.Prenom} {elt.Nom} {elt.Age}");
    }
}
```

Entity Framework

Par défaut

- Toutes les propriétés publiques ayant un getter et un setter seront incluses dans le modèle.
- Toutes les colonnes porteront le nom de la propriété.
- ...

Entity Framework

Par défaut

- Toutes les propriétés publiques ayant un getter et un setter seront incluses dans le modèle.
- Toutes les colonnes porteront le nom de la propriété.
- ...

Remarque

Ce comportement par défaut de **EF** peut être modifié.

Entity Framework

Deux solutions possibles

- **Data annotations** (utilisation de décorateurs)
- **Fluent API** (implémentation de méthodes).

Entity Framework

Ajoutons quelques décorateurs à l'entité `Personne`

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace CoursCodeFirst.Models
{
    [Index(nameof(Nom), nameof(Prenom), IsUnique = true)]
    public class Personne
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int Id { get; set; }

        [MaxLength(20), MinLength(2)]
        public string Nom { get; set; }

        [Required]
        public string Prenom { get; set; }

        public int Age { get; set; }
    }
}
```

Entity Framework

Explication

- `[Required]` : précise que cette colonne ne prend pas la valeur `null`.
- `[Key]` : indique que cet attribut correspond à la clé primaire de la table.
- `[MaxLength(20), MinLength(2)]` : indique la longueur max et min de la chaîne de caractère.
- `[Index(nameof(Nom), nameof(Prenom), IsUnique = true)]` : permet de déclarer un index sur la paire de colonnes (`Nom`, `Prenom`) et de les déclarer uniques.

Entity Framework

`[DatabaseGenerated()]` accepte 3 valeurs

- `DatabaseGeneratedOption.None` : précise qu'il faut préciser la valeur de la clé à l'insertion et que le SGBD ne génère pas la valeur.
- `DatabaseGeneratedOption.Identity` (par défaut) : indique que la valeur de la clé primaire est auto-incrémentale.
- `DatabaseGeneratedOption.Computed` : indique que la valeur de la clé primaire est calculée (par une fonction, `GETDATE()` de **SQL Server** par exemple).

Entity Framework

Autres décorateurs

- `[Column("nomColonne")]` : permet d'avoir un nom de colonne différent de celui de l'attribut
- `[Table("nomTable")]` : indique que la table correspondante à cette entité sera nommée `nomTable`
- `[NotMapped]` : indique que l'attribut sera utilisé seulement niveau applicatif et qu'il n'aura pas de correspondance dans la base de données
- `[ForeignKey("nomColonneTableOrigine")]` : indique que cet attribut provient d'une autre table et fait référence au nom de la clé étrangère
- `[Display(Name = "Valeur à afficher")]` : indique la valeur à afficher lorsqu'on fait référence à cet attribut dans un formulaire par exemple
- `[DataType(DataType.Type)]` : précise un type spécifique d'un attribut : `Password`, `Date`...
- `[Order]` : doit être utilisé avec `[Key]` lorsque la clé primaire est composée
- `[StringLength]` : spécifie la longueur exacte d'une propriété de type `string` ou `array`
- ...

Entity Framework

Remarques

- Toutes les propriétés ayant des types de valeur **.NET** non nullable (comme `int`, `decimal`, `bool`...) sont configurées par **EF** comme obligatoires.
- Toutes les propriétés ayant des types de valeurs .NET pouvant accepter la valeur `Null` (`int?`, `decimal?`, `bool?`...) sont configurées comme facultatives.
- **Entity Framework** n'effectue aucune validation de précision, de mise à l'échelle ou de longueur maximale ou minimale avant de transmettre des données au **SGBD**.

Entity Framework

Après chaque modification, il faut penser à ajouter une migration

```
Add-Migration ModificationPersonne
```

Entity Framework

Après chaque modification, il faut penser à ajouter une migration

```
Add-Migration ModificationPersonne
```

Pour exécuter la migration (et appliquer les changements), exécutez

```
Update-Database
```

Entity Framework

Exécutez le code suivant deux fois et vérifiez qu'il ne passe pas la deuxième fois à cause de la contrainte d'unicité sur le couple `(Nom, Prenom)`

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
         Nom = "Pradel",
         Prenom = "Jacques",
         Age = 45
    };

    context.Personnes.Add(personne);
    context.SaveChanges();

    var personnes = context.Personnes.ToList();
    foreach(var elt in personnes)
    {
        Console.WriteLine($"Bonjour { elt.Prenom } { elt.Nom }");
    }
}
```

Entity Framework

Fluent API

- API **Entity Framework** permettant d'assurer le mapping entre le modèle objet le modèle relationnel.
- Fournissant des méthodes de configurations pour mapper le modèle, définir les clés primaires, clés étrangères...

Entity Framework

Exemples avec Fluent API (à ajouter comme méthode dans `ApplicationContext`)

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Personne>(x =>
    {
        x.ToTable("Personnes")
         .HasKey(p => p.Id);

        x.Property(p => p.Nom)
         .HasColumnName("LastName")
         .HasDefaultValue("Doe");

        x.Property(p => p.Prenom)
         .HasMaxLength(30)
         .HasColumnOrder(3);
    });
}
```

Entity Framework

Autres méthodes de configuration de propriété

- `IsRequired`
- `IsFixedLength`
- `HasColumnType`
- ...

Entity Framework

Question

Si on avait plusieurs classes, devrions nous tout déclarer dans `DbContext` ?

Entity Framework

Question

Si on avait plusieurs classes, devrions nous tout déclarer dans `DbContext` ?

Réponse

Non, on peut utiliser une classe de validation et la déclarer dans `DbContext`.

Entity Framework

Après chaque modification, il faut penser à ajouter une migration

```
Add-Migration ModificationAvecFluentAPI
```

Entity Framework

Après chaque modification, il faut penser à ajouter une migration

```
Add-Migration ModificationAvecFluentAPI
```

Pour exécuter la migration (et appliquer les changements), exécutez

```
Update-Database
```

Entity Framework

Exécutez le code suivant et vérifiez que la valeur par défaut `Doe` a été affectée au tuple inséré

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
         Prenom = "William",
         Age = 45
    };

    context.Personnes.Add(personne);
    context.SaveChanges();

    var personnes = context.Personnes.ToList();
    foreach(var elt in personnes)
    {
        Console.WriteLine($"Bonjour { elt.Prenom } { elt.Nom }");
    }
}
```

Entity Framework

Commençons par créer une classe `PersonneValidator` dans un répertoire `validators`

```
namespace TestEF.Validators
{
    internal class PersonneValidator
    {
    }
}
```

Entity Framework

La classe `PersonneValidator` **doit implémenter l'interface de** `IEntityTypeConfiguration`

```
using MapperClass.Model;
using System.Data.Entity.ModelConfiguration;

namespace TestEF.Validators
{
    internal class PersonneValidator : IEntityTypeConfiguration<Personne>
    {
    }
}
```

Entity Framework

Implémentons la méthode de configuration de l'entité

```
namespace TestEF.Validators
{
    internal class PersonneValidator : IEntityTypeConfiguration<Personne>
    {
        public void Configure(EntityTypeBuilder<Personne> builder)
        {
            builder.ToTable("Personnes")
                  .HasKey(p => p.Id);

            builder.Property(p => p.Nom)
              .HasColumnName("LastName")
              .HasDefaultValue("Doe");

            builder.Property(p => p.Prenom)
              .HasMaxLength(30)
              .HasColumnOrder(3);
        }
    }
}
```

Entity Framework

Déclarons `PersonneValidator` **dans** `ApplicationContext`

```
namespace TestEF
{
    internal class ApplicationContext: DbContext
    {
        public DbSet<Personne> Personnes { get; set; }
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=Formation;
              Trusted_Connection=True;");
        }
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            new PersonneValidator().Configure(modelBuilder.Entity<Personne>());
        }
    }
}
```

Entity Framework

Après chaque modification, il faut penser à ajouter une migration

```
Add-Migration ModificationAvecValidator
```

Entity Framework

Après chaque modification, il faut penser à ajouter une migration

```
Add-Migration ModificationAvecValidator
```

Pour exécuter la migration (et appliquer les changements), exécutez

```
Update-Database
```

Entity Framework

Exécutez le code suivant et vérifiez que la valeur par défaut `Doe` a été affectée au tuple inséré

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
         Prenom = "Sophie",
         Age = 35
    };

    context.Personnes.Add(personne);
    context.SaveChanges();

    var personnes = context.Personnes.ToList();
    foreach(var elt in personnes)
    {
        Console.WriteLine($"Bonjour { elt.Prenom } { elt.Nom }");
    }
}
```

Entity Framework

Pour activer la journalisation et rediriger la sortie vers la console

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=Formation;
      Trusted_Connection=True;");
    optionsBuilder.LogTo(Console.WriteLine)
}
```

Entity Framework

On peut aussi spécifier le niveau de journalisation (Ici `Information`)

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=Formation;
      Trusted_Connection=True;");
    optionsBuilder.LogTo(Console.WriteLine, LogLevel.Information)
}
```

Entity Framework

On peut aussi spécifier le niveau de journalisation (Ici `Information`)

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=Formation;
      Trusted_Connection=True;");
    optionsBuilder.LogTo(Console.WriteLine, LogLevel.Information)
}
```

Les valeurs de l'énumération `LogLevel`

- `None` 6
- `Critical` 5
- `Error` 4
- `Warning` 3
- `Information` 2
- `Debug` 1 (par défaut)
- `Trace` 0

Entity Framework

Transactions

- Par défaut, **Entity Framework** exécute les requêtes en mode transaction.
- En effet, pour chaque appel de la méthode `SaveChanges()`, **Entity Framework** démarre une transaction, exécute toutes les requêtes et valide la transaction.
- Pour un multiple appel de la méthode `SaveChanges()`, il y aura autant de transactions que d'appels.
- **Entity Framework** nous laisse la possibilité de gérer explicitement les transactions.

Entity Framework

Pour activer la journalisation des transactions

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=Formation;
      Trusted_Connection=True;");
    optionsBuilder.LogTo(Console.WriteLine, (eventId, logLevel) => logLevel >= LogLevel.
      Information
                          || eventId == RelationalEventId.ConnectionOpened
                          || eventId == RelationalEventId.TransactionCommitted);
}
```

Entity Framework

Exécutez le code suivant et vérifiez la présence de deux transactions dans la console

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Nom = "Wick",
        Prenom = "John",
        Age = 45
    };
    context.Personnes.Add(personne);
    context.SaveChanges();

    var personne2 = new Personne()
    {
        Nom = "Walker",
        Prenom = "Penny",
        Age = 50
    };
    context.Personnes.Add(personne2);
    context.SaveChanges();

    var personne3 = new Personne()
    {
        Nom = "Green",
        Prenom = "Bill",
        Age = 55
    };
    context.Personnes.Add(personne3);
    context.SaveChanges();
}
```

Entity Framework

Remarques

- Lancez le code précédent et vérifiez la présence de la chaîne suivante : `SET IMPLICIT_TRANSACTIONS OFF`.
- Quand la valeur de `IMPLICIT_TRANSACTIONS` est définie à `OFF`, chaque requête **SQL** est délimitée par une instruction `BEGIN TRANSACTION` invisible et une instruction `COMMIT TRANSACTION` invisible.

Utilisons explicitement les transactions et vérifions la présence d'une seule transaction

```
using (var context = new ApplicationContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            var personne = new Personne()
            {
                Nom = "Wick",
                Prenom = "John",
                Age = 45
            };
            context.Personnes.Add(personne);
            context.SaveChanges();
            var personne3 = new Personne()
            {
                Nom = "Green",
                Prenom = "Bill",
                Age = 55
            };
            context.Personnes.Add(personne3);
            context.SaveChanges();
            transaction.Commit();
        }
        catch (Exception ex)
        {
            transaction.Rollback();
            Console.WriteLine(ex.Message);
        }
    }
}
```

Entity Framework

Remarque

Lancez le code précédent et vérifiez la présence de la chaîne suivante à la fin : `Committed transaction.`.

Entity Framework

Considérons une entité `Adresse` avec une clé primaire composée

```
[PrimaryKey(nameof(Rue), nameof(Ville), nameof(CodeP))]
internal class Adresse
{

    public int Rue { get; set; }

    public string Ville { get; set; }

    public string CodeP { get; set; }
}
```

Entity Framework

Considérons une entité `Adresse` avec une clé primaire composée

```
[PrimaryKey(nameof(Rue), nameof(Ville), nameof(CodeP))]
internal class Adresse
{

    public int Rue { get; set; }

    public string Ville { get; set; }

    public string CodeP { get; set; }
}
```

Le décorateur `[PrimaryKey]` est disponible depuis **EF Core 7**, dans les versions ultérieures, il fallait utiliser **Fluent API**.

Entity Framework

Définissons la relation entre `Personne` **et** `Adresse`

```
[PrimaryKey(nameof(Rue), nameof(Ville), nameof(CodeP))]
internal class Adresse
{

    public int Rue { get; set; }

    public string Ville { get; set; }

    public string CodeP { get; set; }

    public Personne Personne { get; set; }
}
```

Entity Framework

Mettons à jour la classe `ApplicationContext.cs`

```
internal class ApplicationContext : DbContext
{
    public DbSet<Personne> Personnes { get; set; }
    public DbSet<Adresse> Adresses { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=Formation;
          Trusted_Connection=True;");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        new PersonneValidator().Configure(modelBuilder.Entity<Personne>());
    }
}
```

Entity Framework

Créons une migration pour ce changement

```
Add-Migration CreationEntiteAdresse
```

Entity Framework

Créons une migration pour ce changement

```
Add-Migration CreationEntiteAdresse
```

Exécutons la migration

```
Update-Database
```

Entity Framework

Pour tester, lançons le `Main` suivant

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Nom = "Linus",
        Prenom = "Benjamin",
        Age = 45
    };

    var adresse = new Adresse
    {
        Rue = 13,
        Ville = "Marseille",
        CodeP = "13013",
        Personne = personne
    };

    // on persiste les données
    context.Adresses.Add(adresse);
    context.SaveChanges();

    // on affiche les personnes
    foreach (var elt in context.Personnes)
    {
        Console.WriteLine($"Bonjour {elt.Prenom} {elt.Nom}");
    }
}
```

Entity Framework

Constats

- Deux tables créées :
  - `Adresses` avec les colonnes `Rue, Ville, CodeP, #PersonneId`
  - `Personnes` avec les colonnes `Id, Nom, Prenom, Age`
- Deux tuples insérés :
  - `(XX, Linus, Benjamin)` dans `Personnes`
  - `(13, Marseille, 13013, XX)` dans `Adresses`

Entity Framework

Pour définir une relation bidirectionnelle entre les entités `Adresse` et `Personne`

```
[PrimaryKey(nameof(Rue), nameof(Ville), nameof(CodeP))]
internal class Adresse
{
    public int Rue { get; set; }

    public string Ville{ get; set; }

    public string CodeP{ get; set; }

    public Personne Personne { get; set; }
}
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration OneToOneBi
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration OneToOneBi
```

Exécutons la migration

```
Update-Database
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration OneToOneBi
```

Exécutons la migration

```
Update-Database
```


Erreur

Dans quelle table faudra t-il placer la clé étrangère ?

Entity Framework

Définissons la clé étrangère dans `Adresse`

```
[PrimaryKey(nameof(Rue), nameof(Ville), nameof(CodeP))]
internal class Adresse
{

    public int Rue { get; set; }

    public string Ville { get; set; }

    public string CodeP { get; set; }

    public int PersonneId { get; set; }
    public Personne Personne { get; set; }
}
```

Entity Framework

Relançons la migration précédente

```
Add-Migration OneToOneBi
```

Entity Framework

Relançons la migration précédente

```
Add-Migration OneToOneBi
```

Exécutons la migration

```
Update-Database
```

Entity Framework

Pour tester, on peut affecter la personne à l'adresse

```
using (var context = new ApplicationContext())
{
    var personne = new Personne
    {
        Nom = "Ford",
        Prenom = "James",
        Age = 45
    };

    var adresse = new Adresse
    {
        Rue = 2,
        Ville = "Marseille",
        CodeP = "13002",
        Personne = personne
    };

    context.Adresses.Add(adresse);
    context.SaveChanges();
}
```

Entity Framework

Ou en affectant l'adresse à la personne

```
using (ApplicationContext context = new ApplicationContext())
{
    Adresse a = new()
    {
        Rue = 3,
        Ville = "Marseille",
        CodePostal = "13006"
    };

    Personne p = new()
    {
        Nom = "Pradel",
        Prenom = "Jacques",
        Age = 66,
        Adresse = a
    };
    context.Personnes.Add(p);
    context.SaveChanges();
}
```

Entity Framework

Considérons le contenu suivant de `Personne`

```
internal class Personne
{

    public int Id { get; set; }

    public string Nom { get; set; }

    public string Prenom { get; set; }

    public int Age { get; set; }

    public Adresse Adresse { get; set; }
}
```

Entity Framework

Et le contenu suivant de `Adresse`

```
internal class Adresse
{
    public int Rue { get; set; }

    public string Ville { get; set; }

    public string CodeP { get; set; }

    public Personne Personne { get; set; }
}
```

Entity Framework

Dans `ApplicationContext`**, commençons par définir une clé composée pour l'entité** `Adresse`

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    new PersonneValidator().Configure(modelBuilder.Entity<Personne>());

    // pour définir une clé primaire composée
    modelBuilder.Entity<Adresse>()
                .HasKey(e => new { e.Rue, e.CodeP, e.Ville });

}
```

Définissons ensuite une association `OneToOne` **entre** `Personne` **et** `Adresse`

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    new PersonneValidator().Configure(modelBuilder.Entity<Personne>());

    // pour définir une clé primaire composée
    modelBuilder.Entity<Adresse>()
                .HasKey(e => new { e.Rue, e.CodeP, e.Ville });

    //
    modelBuilder.Entity<Personne>()
                .HasOne(e => e.Adresse)
                .WithOne(e => e.Personne)
                .HasForeignKey<Adresse>();
}
```

Définissons ensuite une association `OneToOne` entre `Personne` et `Adresse`

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    new PersonneValidator().Configure(modelBuilder.Entity<Personne>());

    // pour définir une clé primaire composée
    modelBuilder.Entity<Adresse>()
                .HasKey(e => new { e.Rue, e.CodeP, e.Ville });

    //
    modelBuilder.Entity<Personne>()
                .HasOne(e => e.Adresse)
                .WithOne(e => e.Personne)
                .HasForeignKey<Adresse>();
}
```

Remarques

- `modelBuilder.Entity<Personne>()` : permet de spécifier l'entité à configurer.
- `.HasOne(e => e.Adresse)` : signifie qu'une personne a une adresse.
- `.WithOne(e => e.Personne)` : signifie qu'une adresse est associée à une personne.
- `hasForeignKey<Adresse>()` : signifie que la clé étrangère sera dans la table `Adresses` et qu'elle référencera la clé primaire de la table `Personnes`.

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration OneToOneFluentAPI
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration OneToOneFluentAPI
```

Exécutons la migration

```
Update-Database
```

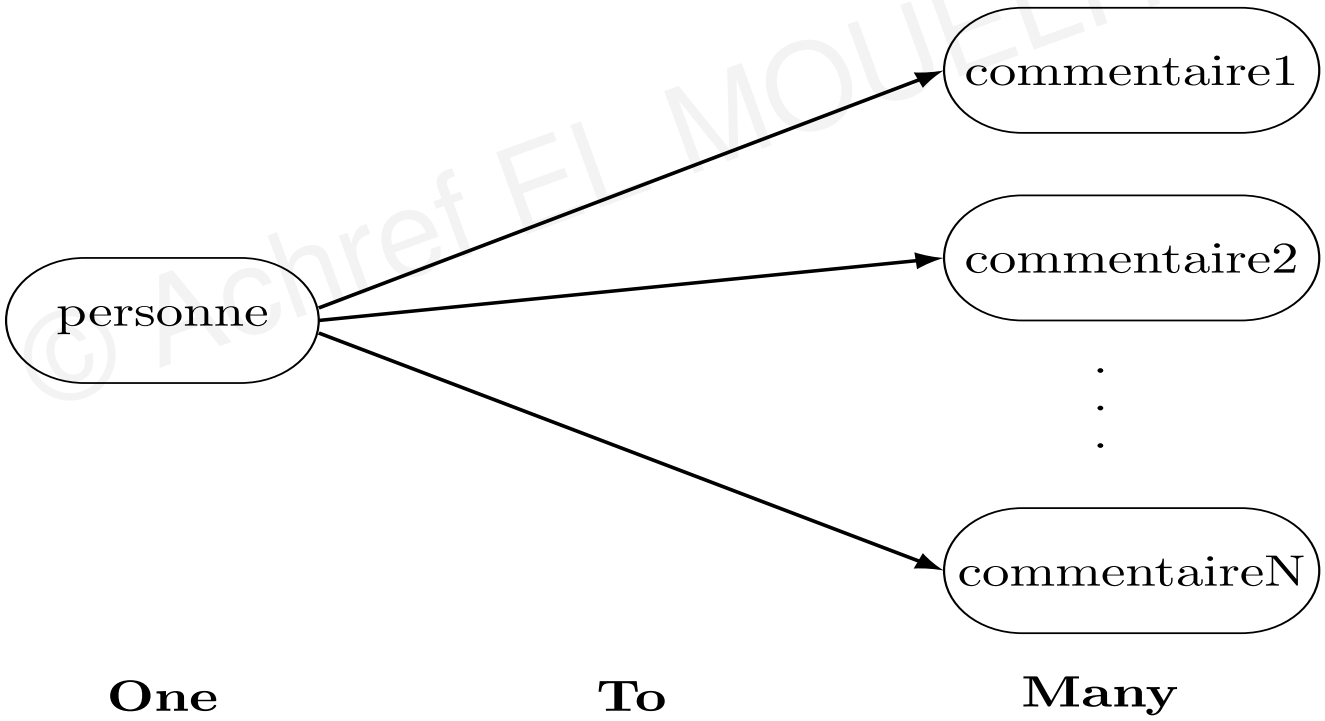
Et pour tester

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Nom = "Ford",
        Prenom = "James",
        Age = 45
    };

    var adresse = new Adresse
    {
        Rue = 2,
        Ville = "Marseille",
        CodeP = "13002",
        Personne = personne
    };

    // on persiste les données
    context.Adresses.Add(adresse);
    context.SaveChanges();

    // on affiche les personnes
    foreach (var elt in context.Personnes)
    {
        Console.WriteLine($"Bonjour {elt.Prenom} {elt.Nom}");
        if (elt.Adresse != null)
        {
            Console.Write($" {elt.Adresse.CodeP} - {elt.Adresse.Ville}");
        }
    }
}
```

Entity Framework

Hypothèse

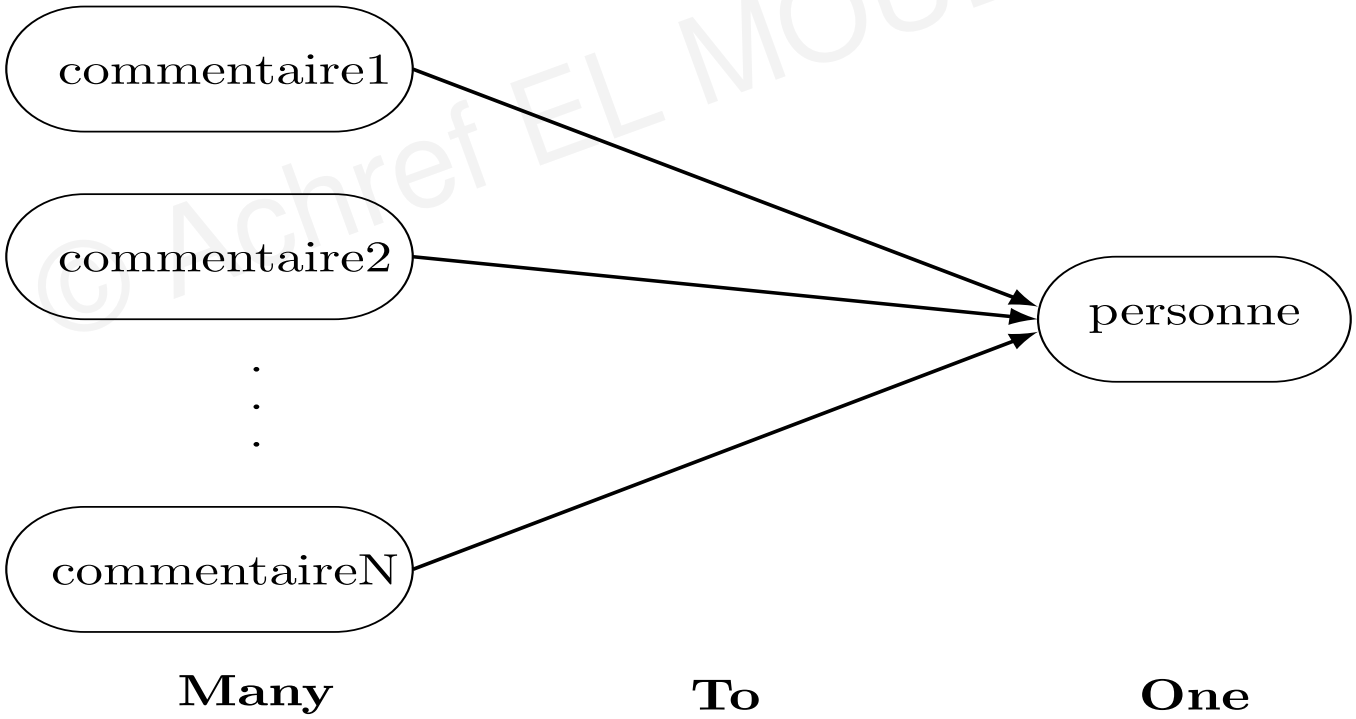
Une personne pourrait rédiger un ou plusieurs commentaires.



Entity Framework

Hypothèse

Si l'entité `Personne` a une relation **OneToMany** avec l'entité `Commentaire`, alors on peut dire que l'entité `Commentaire` a une relation **ManyToOne** avec l'entité `Personne`



Entity Framework

Créons l'entité `Commentaire`

```
internal class Commentaire
{

    public int Id { get; set; }

    public string Titre { get; set; }

    public string Contenu { get; set; }

}
```

Entity Framework

Intégrons `Commentaire` **dans** `Personne`

```
internal class Personne
{

    public int Id { get; set; }

    public string Nom { get; set; }

    public string Prenom { get; set; }

    public int Age { get; set; }

    public IList<Commentaire> Commentaires { get; set; }

}
```

Entity Framework

Et `Personne` **dans** `Commentaire`

```
internal class Commentaire
{

    public int Id { get; set; }

    public string Titre { get; set; }

    public string Contenu { get; set; }

    public int PersonneId { get; set; }

    public Personne Personne { get; set; }

}
```

Entity Framework

Mettons à jour la classe `ApplicationContext.cs`

```
internal class ApplicationContext : DbContext
{
    public DbSet<Personne> Personnes { get; set; }
    public DbSet<Adresse> Adresses { get; set; }
    public DbSet<Commentaire> Commentaires { get; set; }

    // + méthodes précédentes

}
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration OneToMany
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration OneToMany
```

Exécutons la migration

```
Update-Database
```

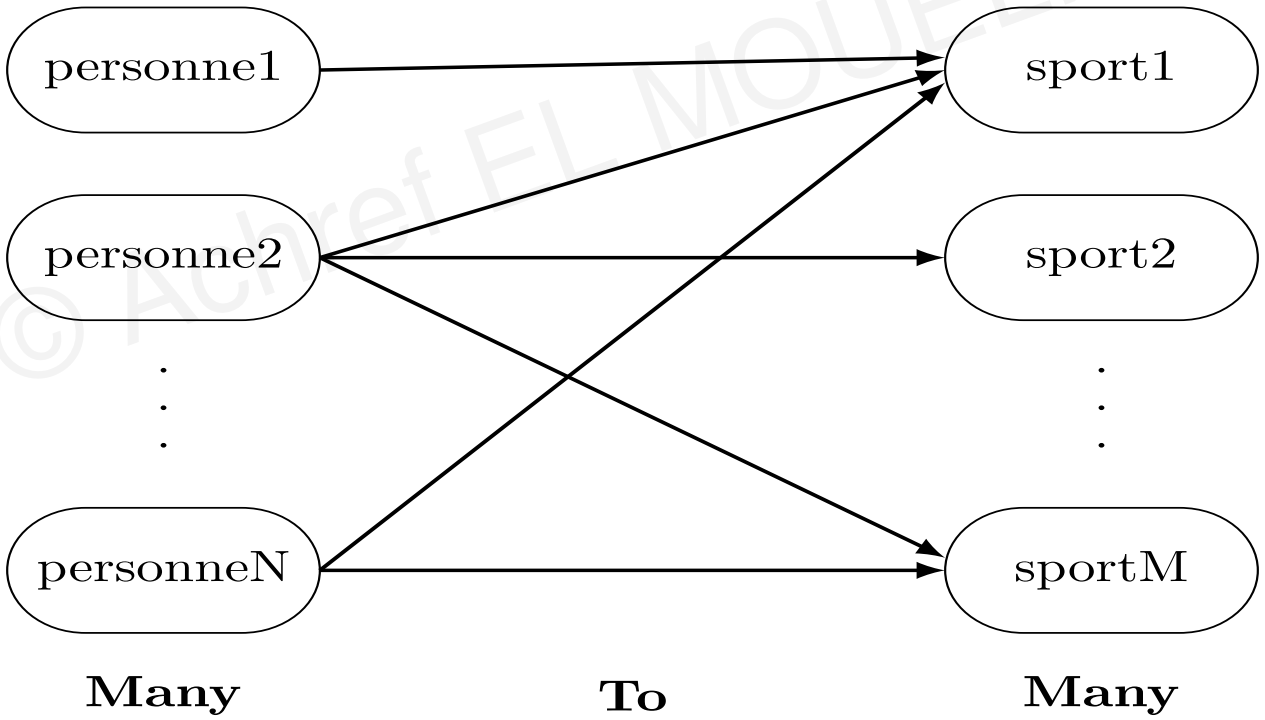
Et pour tester

```
using (var context = new ApplicationContext())
{
    var commentaire1 = new Commentaire
    {
        Titre = "C#",
        Contenu = "C# est un langage Microsoft"
    };
    var commentaire2 = new Commentaire
    {
        Titre = "Java",
        Contenu = "Java est un langage Oracle"
    };
    var personne = new Personne()
    {
        Nom = "Tancredi",
        Prenom = "Sara",
        Age = 40,
        Commentaires = new List<Commentaire> { commentaire1, commentaire2 }
    };
    // on persiste les données
    context.Personnes.Add(personne);
    context.SaveChanges();
    // on affiche les personnes
    foreach (var elt in context.Personnes.Include(p => p.Commentaires))
    {
        Console.WriteLine($"{elt.Prenom} {elt.Nom} :");
        foreach(var c in elt.Commentaires)
        {
            Console.WriteLine($"{c.Titre} - {c.Contenu}");
        }
    }
}
```

Entity Framework

Exemple

- Une personne peut pratiquer plusieurs sports
- Un sport peut être pratiqué par plusieurs personnes



Entity Framework

Commençons par créer l'énumération `SportType` suivante dans `Enums`

```
internal enum SportType
{
    COLLECTIF,
    INDIVIDUEL
}
```

Entity Framework

Créons aussi une entité `Sport`

```
internal class Sport
{
    public int Id { get; set; }

    public string Nom { get; set; }

    [EnumDataType(typeof(SportType))]
    public SportType TypeDeSport { get; set; }

}
```

Entity Framework

Intégrons `Sport` **dans** `Personne`

```
internal class Personne
{

    public int Id { get; set; }

    public string Nom { get; set; }

    public string Prenom { get; set; }

    public int Age { get; set; }

    public Adresse Adresse { get; set; }

    public IList<Commentaire> Commentaires { get; set; }

    public IList<Sport>? Sports { get; set; }

}
```

Entity Framework

Et intégrons `Personne` **dans** `Sport`

```
internal class Sport
{
    public int Id { get; set; }

    public string Nom { get; set; }

    [EnumDataType(typeof(SportType))]
    public SportType TypeDeSport { get; set; }

    public IList<Personne> Personnes { get; set; }

}
```

Entity Framework

Mettons à jour la classe `ApplicationContext.cs`

```
internal class ApplicationContext : DbContext
{
    public DbSet<Personne> Personnes { get; set; }
    public DbSet<Adresse> Adresses { get; set; }
    public DbSet<Commentaire> Commentaires { get; set; }
    public DbSet<Sport> Sports { get; set; }

    // + méthodes précédentes

}
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration ManyToMany
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration ManyToMany
```

Exécutons la migration

```
Update-Database
```

Et pour tester

```
using (var context = new ApplicationContext())
{
    Sport sport1 = new()
    {
        Nom = "Hand",
        TypeDeSport = SportType.COLLECTIF
    };
    Sport sport2 = new()
    {
        Nom = "Natation",
        TypeDeSport = SportType.INDIVIDUEL
    };
    var personne = new Personne()
    {
        Nom = "Lock",
        Prenom = "John",
        Age = 60,
        Sports = [sport1, sport2 ]
    };

    context.Personnes.Add(personne);
    context.SaveChanges();

    foreach (var elt in context.Personnes.Include(p => p.Sports))
    {
        Console.WriteLine($"{elt.Prenom} {elt.Nom} :");
        foreach (var s in elt.Sports)
        {
            Console.WriteLine($"{s.Nom} - {s.TypeDeSport}");
        }
    }
}
```

Entity Framework

Étant donné l'exemple suivant

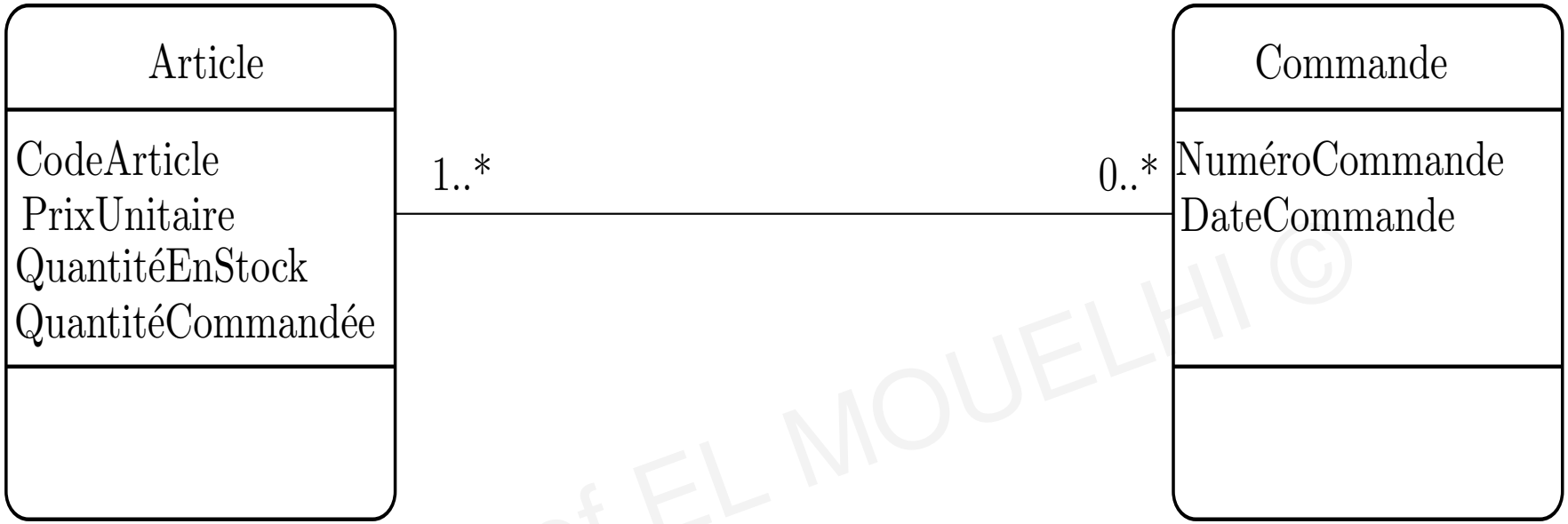
Entity Framework

Étant donné l'exemple suivant

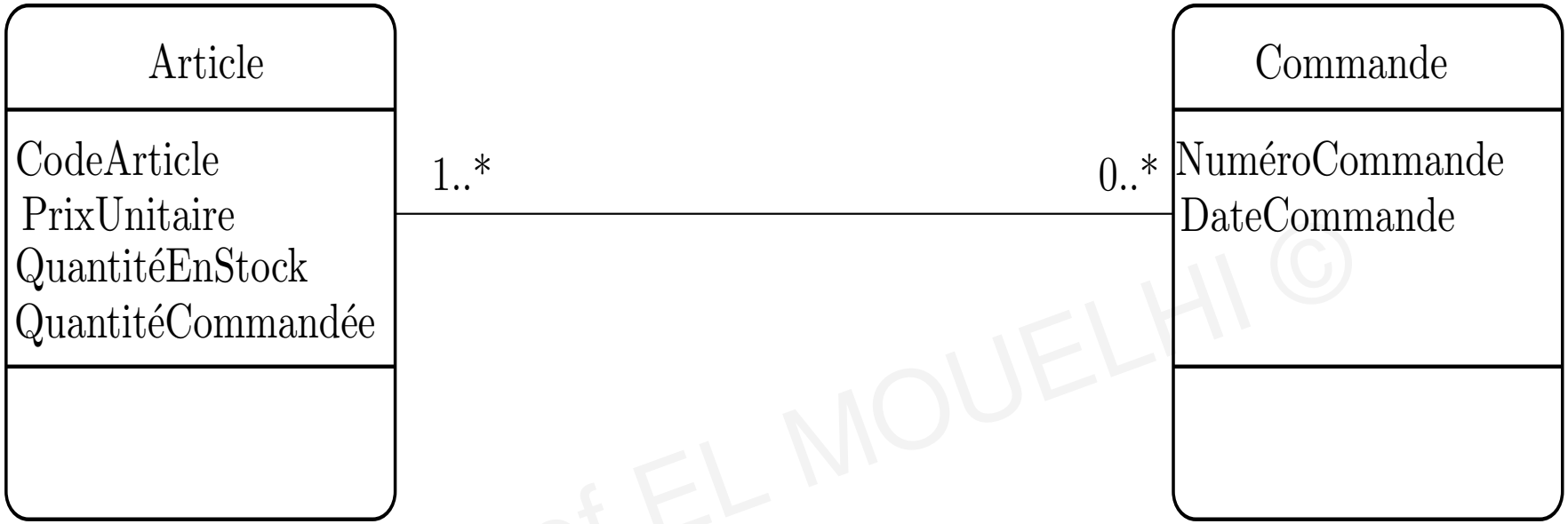
Question

Où peut-on placer la `quantitéCommandée` ?

Dans `Article` ?



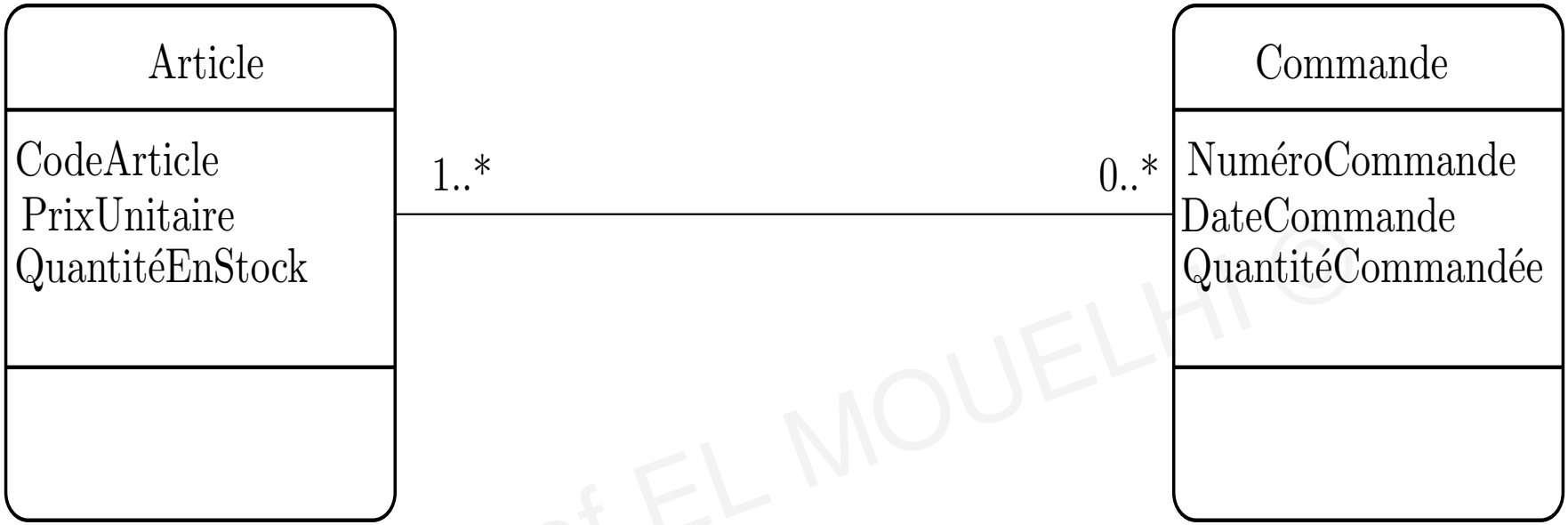
Dans `Article` ?



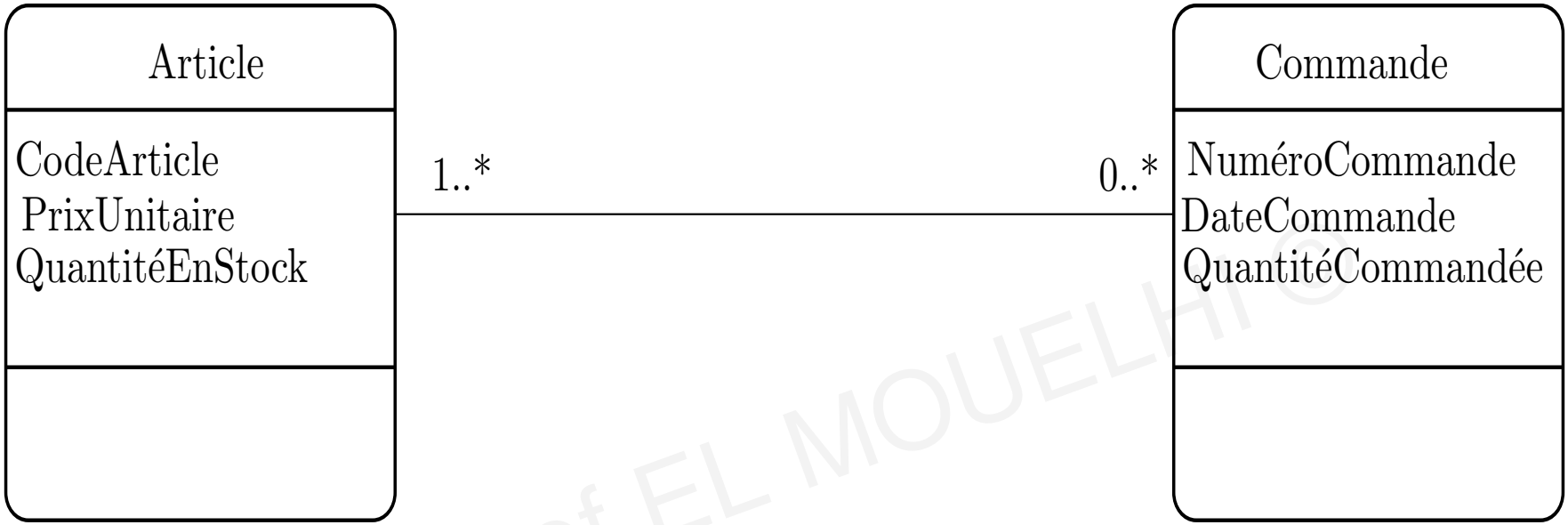
Impossible

Ainsi, une seule `quantitéCommandée` pour tous ceux qui commandent le même article.

Dans `Commande` ?



Dans `Commande` ?



Impossible aussi

Ainsi, une seule `quantitéCommandée` pour tous les articles d'une même commande.

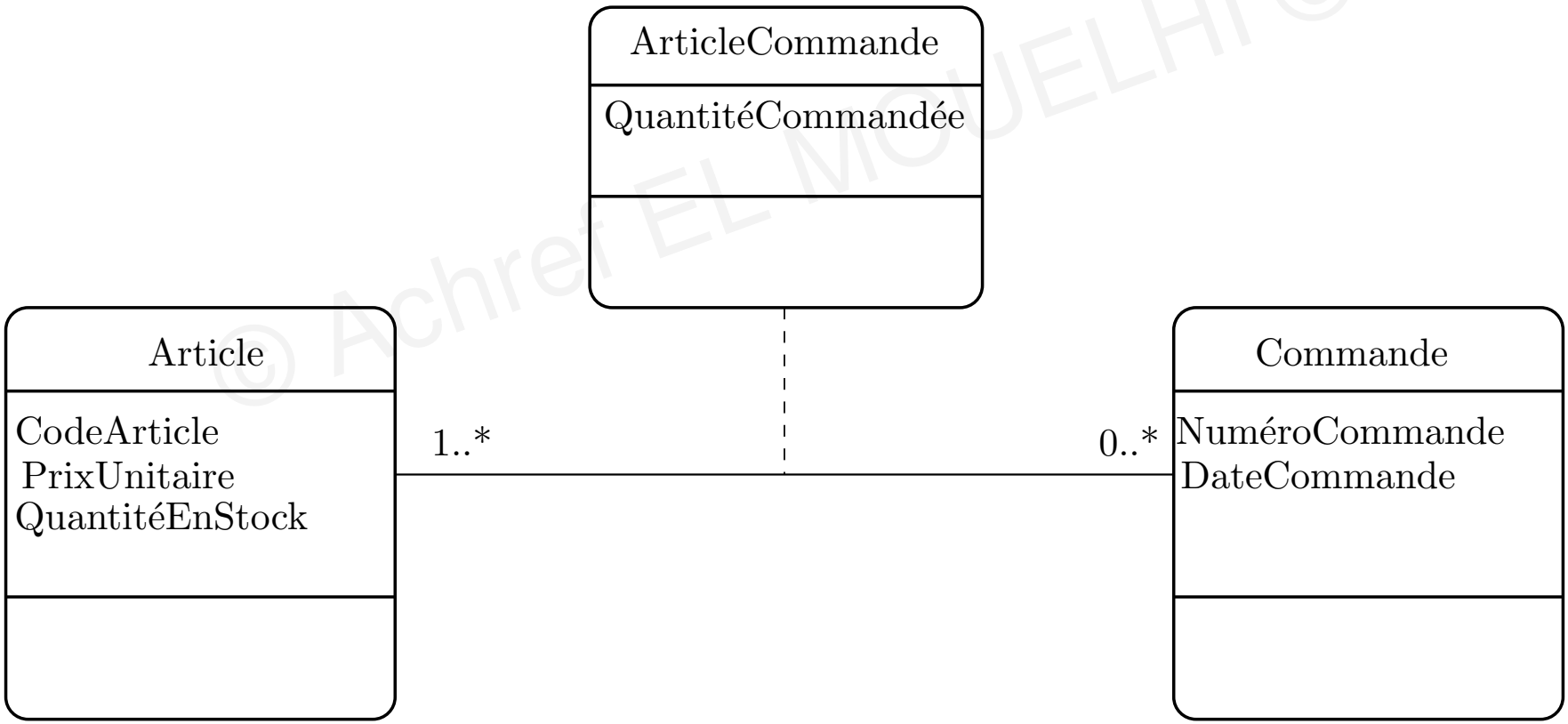
Conclusion

La `quantitéCommandée` ne peut être ni dans `Article` ni dans `Commande`.

Conclusion

La `quantitéCommandée` ne peut être ni dans `Article` ni dans `Commande`.

Solution : créer une classe d'association



Entity Framework

Si la classe association est porteuse de données

- Par exemple : la relation (ArticleCommande) entre Commande et Article
- Il faut préciser la quantité de chaque article dans une commande

Entity Framework

Si la classe association est porteuse de données

- Par exemple : la relation (ArticleCommande) entre Commande et Article
- Il faut préciser la quantité de chaque article dans une commande

Solution

- Créer trois entités `Article`, `Commande` et `ArticleCommande`
- Dans `Article`, définir une relation `OneToMany` avec `ArticleCommande`.
- Dans `Commande`, définir une relation `OneToMany` avec `ArticleCommande`.
- Dans `ArticleCommande`, définir deux relations `ManyToOne` une avec `Commande` et une avec `Article`.

Entity Framework

Lazy loading : chargement paresseux

Par défaut, **Entity Framework**, comme la plupart des **ORM**, ne charge pas les entités liées (dépendantes).

Entity Framework

Lazy loading : chargement paresseux

Par défaut, **Entity Framework**, comme la plupart des **ORM**, ne charge pas les entités liées (dépendantes).

Le code suivant génère un `NullReferenceException`

```
using (var context = new ApplicationContext())
{
    var sports = context.Personnes.First(elt => elt.Id == 3).Sports;
    foreach (var item in sports)
    {
        Console.WriteLine($"{item.TypeDeSport} - {item.Nom}");
    }
}
```

Entity Framework

Pour charger l'entité dépendante au même moment que l'entité principale, on peut utiliser `include` [Eager loading ou chargement anticipé]

```
using (var context = new ApplicationContext())
{
    var sports = context.Personnes.Include(elt => elt.Sports).First(elt => elt.Id == 3).Sports;
    foreach (var item in sports)
    {
        Console.WriteLine($"{item.TypeDeSport} - {item.Nom}");
    }
}
```

Entity Framework

Exercice

En utilisant **Entity Framework**, écrire un code qui permet d'insérer une personne dans la base de données avec deux sports

- le premier : il existe déjà dans la base de données
- le deuxième : c'est un nouveau

Solution

```
using (var context = new ApplicationContext())
{
    var newSport = new Sport()
    {
        Nom = "Basket",
        TypeDeSport = SportType.COLLECTIF
    };

    var oldSport = context.Sports.Where(s => s.Nom.Equals("Natation")).FirstOrDefault();

    var personne = new Personne
    {
        Nom = "Parker",
        Prenom = "Tony",
        Age = 45,
        Sports = new List<Sport>() { newSport, oldSport }
    };

    context.Personnes.Add(personne);
    context.SaveChanges();

    var jordan = context.Personnes
        .Include(p => p.Sports)
        .Where(p => p.Nom.Equals("Parker"))
        .FirstOrDefault();

    foreach (var elt in jordan.Sports)
    {
        Console.WriteLine($"{elt.TypeDeSport} : {elt.Nom}");
    }

}
```

Entity Framework

Supposant que l'on a

- une classe mère `Personne`
- deux classes filles `Etudiant` et `Enseignant` qui étendent `Personne`

Entity Framework

Comment transformer l'association d'héritage en tables ? (3 méthodes différentes)

- Une seule table pour les trois classes : **Table per Hierarchy (TPH)** [par défaut]
- Une table indépendante pour chaque classe : **Table per Type (TPT)**
- Une table pour chaque classe concrète : **Table per Concrete Class (TPC)** [EF Core 7]

Entity Framework

Comment transformer l'association d'héritage en tables ? (3 méthodes différentes)

- Une seule table pour les trois classes : **Table per Hierarchy (TPH)** [par défaut]
- Une table indépendante pour chaque classe : **Table per Type (TPT)**
- Une table pour chaque classe concrète : **Table per Concrete Class (TPC)** [EF Core 7]

Remarque

Considérons un projet pour chaque méthode de transformation d'héritage (dans la même solution).

Entity Framework

Considérons le contenu suivant pour l'entité `Personne`

```
internal class Personne
{

    public int Id { get; set; }

    public string Nom { get; set; }

    public string Prenom { get; set; }

    public int Age { get; set; }

}
```

Entity Framework

Créons l'entité `Etudiant`

```
internal class Etudiant: Personne
{
    public int Bourse { get; set; }
}
```

Entity Framework

Créons l'entité `Etudiant`

```
internal class Etudiant: Personne
{
    public int Bourse { get; set; }
}
```

Et l'entité `Enseignant`

```
internal class Enseignant: Personne
{
    public int Salaire { get; set; }
}
```

Entity Framework

Déclarons les nouvelles entités dans `ApplicationContext`

```
internal class ApplicationContext : DbContext
{
    public DbSet<Personne> Personnes { get; set; }
    public DbSet<Etudiant> Etudiants { get; set; }
    public DbSet<Enseignant> Enseignants { get; set; }

    // + méthodes précédentes

}
```

Entity Framework

Dans `ApplicationContext`, utilisons Fluent API pour définir la colonne de discrimination et ses valeurs (une pour chaque entité)

```
protected override void OnModelCreating(ModelBuilder
   modelBuilder)
{
    modelBuilder.Entity<Personne>()
        .HasDiscriminator<string>("type_pers")
        .HasValue<Enseignant>("ens")
        .HasValue<Etudiant>("etu")
        .HasValue<Personne>("pers");

    // + code précédent

}
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration TPH
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration TPH
```

Exécutons la migration

```
Update-Database
```

Dans le `Main`, on prépare les objets à persister

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Nom = "Pradel",
        Prenom = "Jacques",
        Age = 45
    };
    context.Personnes.Add(personne);
    var etudiant = new Etudiant()
    {
        Nom = "Mary",
        Prenom = "Michel",
        Age = 55,
        Bourse = 500
    };
    context.Etudiants.Add(etudiant);
    var enseignant = new Enseignant()
    {
        Nom = "Hanin",
        Prenom = "Roger",
        Age = 65,
        Salaire = 6000
    };
    context.Enseignants.Add(enseignant);
    context.SaveChanges();

    foreach (var elt in context.Personnes)
    {
        Console.WriteLine($"Bonjour {elt.Prenom} {elt.Nom}");
    }
}
```

Entity Framework

Allons voir la base de données

- Une seule table `Personnes` a été créée
- Les colonnes `Id`, `Nom`, `Prenom`, `Age`, `Salaire`, `Bourse` et `type_pers`
- La personne `Jacques Pradel` a la valeur `null` dans `Salaire` et `Bourse` et la valeur `pers` dans `type_pers`
- L'étudiant `Michel Mary` a la valeur `null` dans `Bourse` et la valeur `etu` dans `type_pers`
- L'enseignant `Roger Hanin` a la valeur `null` dans `Salaire` et la valeur `ens` dans `type_pers`

Pour récupérer et afficher les étudiants

```
using (var context = new ApplicationContext())
{
    foreach (var e in context.Etudiants)
    {
        Console.WriteLine($"Bonjour {e.Prenom} {e.Nom} {e.Bourse}");
    }
    // affiche Bonjour Michel Mary 500
}
```

Pour récupérer et afficher les étudiants

```
using (var context = new ApplicationContext())
{
    foreach (var e in context.Etudiants)
    {
        Console.WriteLine($"Bonjour {e.Prenom} {e.Nom} {e.Bourse}");
    }
    // affiche Bonjour Michel Mary 500
}
```

Pour récupérer et afficher les enseignants

```
using (var context = new ApplicationContext())
{
    foreach (var e in context.Enseignants)
    {
        Console.WriteLine($"Bonjour {e.Prenom} {e.Nom} {e.Salaire}");
    }
    // affiche Bonjour Roger Hanin 6000
}
```

Entity Framework

Exercice

Écrire un code qui permet de sélectionner les personnes qui ne sont ni enseignants ni étudiants.

Entity Framework

Table per Concrete Class : commençons par créer l'entité `Personne`

```
public abstract class Personne
{

    public int Id { get; set; }

    public string Nom { get; set; }

    public string Prenom { get; set; }

    public int Age { get; set; }

}
```

Entity Framework

Créons l'entité `Etudiant`

```
public class Etudiant: Personne
{
    public int Bourse { get; set; }
}
```

Et l'entité `Enseignant`

```
public class Enseignant: Personne
{
    public int Salaire { get; set; }
}
```

Entity Framework

Déclarons uniquement l'entité `Personne` dans `ApplicationContext`

```
internal class ApplicationContext : DbContext
{
    public DbSet<Personne> Personnes { get; set; }


    // + méthodes précédentes

}
```

Entity Framework

Dans `ApplicationContext`, utilisons Fluent API pour déclarer la TPC

```
protected override void OnModelCreating(ModelBuilder
   modelBuilder)
{

    modelBuilder.Entity<Personne>().UseTpcMappingStrategy();
    modelBuilder.Entity<Etudiant>().ToTable("Etudiants");
    modelBuilder.Entity<Enseignant>().ToTable("Enseignants");

    // + code précédent

}
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration TPC
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration TPC
```

Exécutons la migration

```
Update-Database
```

Dans `Main`, on prépare les objets et on les persiste

```
using (var context = new ApplicationContext())
{
    var etudiant = new Etudiant()
    {
        Id = 1,
        Nom = "Mary",
        Prenom = "Michel",
        Age = 35,
        Bourse = 500
    };
    context.Personnes.Add(etudiant);
    var enseignant = new Enseignant()
    {
        Id = 2,
        Nom = "Hanin",
        Prenom = "Roger",
        Age = 65,
        Salaire = 6000
    };
    context.Personnes.Add(enseignant);
    context.SaveChanges();
}
```

Entity Framework

Allons voir la base de données

- deux tables `Etudiant` et `Enseignant` ont été créées.
- la table `Etudiant` a les colonnes `Id`, `Nom`, `Prenom`, `Age` et `Bourse`.
- la table `Enseignant` a les colonnes `Id`, `Nom`, `Prenom`, `Age` et `Salaire`.

Dans `Main`, pour récupérer les étudiants et les enseignants

```
using (var context = new ApplicationContext())
{
    var etudiants = context.Personnes
        .OfType<Etudiant>()
        .ToList();

    foreach (var e in etudiants)
    {
        Console.WriteLine($"{e.Nom} - {e.Bourse}");
        // Mary - 500
    }
}
```

Dans `Main`, pour récupérer les étudiants et les enseignants

```
using (var context = new ApplicationContext())
{
    var etudiants = context.Personnes
       .OfType<Etudiant>()
       .ToList();

    foreach (var e in etudiants)
    {
        Console.WriteLine($"{e.Nom} - {e.Bourse}");
        // Mary - 500
    }
}
```

Et pour récupérer seulement les étudiants (ou les enseignants)

```
using (var context = new ApplicationContext())
{
    var enseignants = context.Personnes
       .OfType<Enseignant>()
       .ToList();

    foreach (var e in enseignants)
    {
        Console.WriteLine($"{e.Nom} - {e.Salaire}");
        // Hanin - 6000
    }
}
```

Entity Framework

Table per Type : commençons par créer l'entité `Personne`

```
internal class Personne
{

    public int Id { get; set; }

    public string Nom { get; set; }

    public string Prenom { get; set; }

    public int Age { get; set; }

}
```

Entity Framework

Créons l'entité `Etudiant`

```
internal class Etudiant: Personne
{
    public int Bourse { get; set; }
}
```

Et l'entité `Enseignant`

```
internal class Enseignant: Personne
{
    public int Salaire { get; set; }
}
```

Entity Framework

Déclarons les nouvelles entités dans `ApplicationContext`

```
internal class ApplicationContext : DbContext
{
    public DbSet<Personne> Personnes { get; set; }
    public DbSet<Etudiant> Etudiants { get; set; }
    public DbSet<Enseignant> Enseignants { get; set; }

    // + méthodes précédentes

}
```

Entity Framework

Dans `ApplicationContext`, utilisons Fluent API pour définir la colonne de discrimination et ses valeurs (une pour chaque entité)

```
protected override void OnModelCreating(ModelBuilder
   modelBuilder)
{
    modelBuilder.Entity<Personne>().UseTptMappingStrategy();

    // + code précédent

}
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration TPT
```

Entity Framework

Ajoutons une nouvelle migration

```
Add-Migration TPT
```

Exécutons la migration

```
Update-Database
```

Dans le `Main`, on prépare les objets à persister

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Nom = "Pradel",
        Prenom = "Jacques",
        Age = 45
    };
    context.Personnes.Add(personne);
    var etudiant = new Etudiant()
    {
        Nom = "Mary",
        Prenom = "Michel",
        Age = 35,
        Bourse = 500
    };
    context.Etudiants.Add(etudiant);
    var enseignant = new Enseignant()
    {
        Nom = "Hanin",
        Prenom = "Roger",
        Age = 65,
        Salaire = 6000
    };
    context.Enseignants.Add(enseignant);
    context.SaveChanges();

    foreach (var elt in context.Personnes)
    {
        Console.WriteLine($"Bonjour {elt.Prenom} {elt.Nom}");
    }
}
```

Entity Framework

Allons voir la base de données

- trois tables `Personne`, `Etudiant` et `Enseignant` ont été créées
- la table `Personne` a les colonnes `Id`, `Nom`, `Prenom` et `Age`
- la table `Etudiant` a les colonnes `Id` et `Bourse`
- la table `Enseignant` a les colonnes `Id` et `Salaire`

Entity Framework

EF Core : deux modes

- Synchrone
- Asynchrone (non bloquant) avec les `Task`

Entity Framework

Exemple avec `FindAsync`

```
using (var context = new ApplicationContext())
{
    var p = await context.Personnes.FindAsync(8);
    Console.WriteLine($"{p.Id} {p.Prenom} {p.Nom} {p.Age}");
}
```

Entity Framework

Exemple avec `SaveChangesAsync`

```
using (var context = new ApplicationContext())
{
    var p = await context.Personnes.FindAsync(8);

    p.Nom = "Doe";

    await context.SaveChangesAsync();

    foreach (var elt in context.Personnes)
    {
        Console.WriteLine($"{elt.Id} {elt.Prenom} {elt.Nom}");
    }
}
```

Entity Framework

Exemple avec `AddAsync`

```
using (var context = new ApplicationContext())
{
    var personne = new Personne()
    {
        Nom = "Wildmore",
        Prenom = "Charles",
        Age = 65
    };
    await context.Personnes.AddAsync(personne);

    await context.SaveChangesAsync();
    foreach (var elt in context.Personnes)
    {
        Console.WriteLine($"{elt.Id} {elt.Prenom} {elt.Nom}");
    }
}
```

Entity Framework

Objectif

- Organiser l'application en couches
- Faciliter la réutilisation du code

Entity Framework

Commençons par définir une interface générique contenant les principales méthodes de lecture/écriture

```
internal interface IRepository<T> where T : class
{

    IQueryable<T> GetAll();

    Task<T> GetByIdAsync(int id);

    Task AddAsync(T entity);

    Task UpdateAsync(T entity);

    Task DeleteAsync(T entity);

}
```

Entity Framework

Créons une classe générique `Repository` implémentant l'interface `IRepository`

```
internal class Repository<T> : IRepository<T> where T : class
{
    private readonly DbContext _context;
    private readonly DbSet<T> _dbSet;

    public Repository()
    {
        _context = new FormationContext();
        _dbSet = _context.Set<T>();
    }
}
```

Implémentons ensuite toutes les méthodes de l'interface `IRepository` dans `Repository`

```
internal class Repository<T> : IRepository<T> where T : class
{

    public IQueryable<T> GetAll()
    {
        return _dbSet.AsQueryable();
    }

    public async Task<T> GetByIdAsync(int id)
    {
        return await _dbSet.FindAsync(id);
    }

    public async Task AddAsync(T entity)
    {
        await _dbSet.AddAsync(entity);
        await _context.SaveChangesAsync();
    }

    public async Task UpdateAsync(T entity)
    {
        _context.Entry(entity).State = EntityState.Modified;
        await _context.SaveChangesAsync();
    }

    public async Task DeleteAsync(T entity)
    {
        _dbSet.Remove(entity);
        await _context.SaveChangesAsync();
    }
}
```

Entity Framework

Pour la couche métier, commençons par l'interface générique `IService`

```
internal interface IService<T> where T : class
{

    Task<IEnumerable<T>> GetAllAsync();

    Task<T> GetByIdAsync(int id);

    Task AddAsync(T entity);

    Task UpdateAsync(T entity);

    Task DeleteAsync(int id);

}
```

Entity Framework

Créons une classe générique `Service` implémentant l'interface `IService`

```
internal class Service<T> : IService<T> where T : class
{
    private readonly IRepository<T> _repository;

    public Service(IRepository<T> repository)
    {
        _repository = repository;
    }

}
```

Implémentons ensuite toutes les méthodes de l'interface `IService` dans `Service`

```
internal class Service<T> : IService<T> where T : class
{

    public async Task<IEnumerable<T>> GetAllAsync()
    {
        return await _repository.GetAll().ToListAsync();
    }

    public async Task<T> GetByIdAsync(int id)
    {
        return await _repository.GetByIdAsync(id);
    }

    public async Task AddAsync(T entity)
    {
        await _repository.AddAsync(entity);
    }

    public async Task UpdateAsync(T entity)
    {
        await _repository.UpdateAsync(entity);
    }

    public async Task DeleteAsync(int id)
    {
        var entityToDelete = await _repository.GetByIdAsync(id);
        if (entityToDelete != null)
        {
            await _repository.DeleteAsync(entityToDelete);
        }
    }
}
```

Entity Framework

Créons enfin la classe `PersonneService`

```
internal class PersonneService : Service<Personne>
{
    public PersonneService() : base(new Repository<Personne>())
    {
    }
}
```

Entity Framework

Pour tester l'ajout et la récupération

```
var personne = new Personne
{
    Age = 38,
    Nom = "El Mouelhi",
    Prenom = "Achref"
};

var personneService = new PersonneService();
await personneService.AddAsync(personne);

var personnes = personneService.GetAllAsync().Result;

foreach (var elt in personnes)
{
    Console.WriteLine($"{elt.Id} {elt.Prenom} {elt.Nom}");
}
```

Entity Framework

Pour tester la suppression

```
var personneService = new PersonneService();

await personneService.DeleteAsync(1);

var personnes = personneService.GetAllAsync().Result;

foreach (var elt in personnes)
{
    Console.WriteLine($"{elt.Id} {elt.Prenom} {elt.Nom}");
}
```

Entity Framework

Et la modification

```
var personne = new Personne
{
    Id = 8,
    Age = 0,
    Nom = "Unknown",
    Prenom = "Unknown"
};

var personneService = new PersonneService();

await personneService.UpdateAsync(personne);

var personnes = personneService.GetAllAsync().Result;

foreach (var elt in personnes)
{
    Console.WriteLine($"{elt.Id} {elt.Prenom} {elt.Nom}");
}
```

Entity Framework

Objectif

Générer le `DBContext` et les entités à partir d'une base de données existante.

Entity Framework

Nouveau projet sous **Visual Studio Community 2022**

- Dans la solution `CoursEntityFramework`, créez un projet Console (`CoursScaffolding`)
- Installez les dépendances (**NuGet**) suivantes
  - **Microsoft.EntityframeworkCore.SqlServer** qui inclut déjà **Microsoft.EntityFrameworkCore**
  - **Microsoft.EntityFrameworkCore.Tools** pour gérer les migrations

Entity Framework

Scaffolding : deux arguments obligatoires

- **Connection string**
- **Provider** : fournisseur de base de données (dans notre cas **Microsoft.EntityFrameworkCore.SqlServer**)

Entity Framework

Scaffolding : arguments optionnels

- `-Tables` permet de lister les tables à transformer en entités
- `-UseDatabaseNames` permet d'avoir le même nom pour les entités que les tables
- `-DataAnnotations` permet de définir la configuration par décorateur pour les attributs d'une entité
- `-Context` permet de personnaliser le nom de la classe contexte. Par défaut, le nom est `DatabaseNameContext`.
- `-OutputDir` permet d'indiquer l'emplacement des classes générées (entités et contexte)
- `-ContextDir` permet d'indiquer l'emplacement de la classe générée pour le contexte

Entity Framework

Exemple

```
Scaffold-DbContext 'Data Source=(localdb)\MSSQLLocalDB;Initial
Catalog=Formation' Microsoft.EntityFrameworkCore.SqlServer
-ContextDir Data -OutputDir Models -DataAnnotations
```

Entity Framework

Exemple

```
Scaffold-DbContext 'Data Source=(localdb)\MSSQLLocalDB;Initial
Catalog=Formation' Microsoft.EntityFrameworkCore.SqlServer
-ContextDir Data -OutputDir Models -DataAnnotations
```

Constats : vérifiez la présence

- d'un répertoire `Models` contenant toutes les entités de la base de données
- d'un répertoire `Data` contenant la classe contexte

Entity Framework

Et pour tester

```
using (var context = new FormationContext())
{
    var personne = new Personne()
    {
        Nom = "Dupont",
        Prenom = "Sophie",
        Age = 35
    };

    context.Personnes.Add(personne);
    context.SaveChanges();

    foreach (var elt in context.Personnes)
    {
        Console.WriteLine($"Bonjour {elt.Prenom} {elt.Nom}");
    }
}
```