TP ProductStore

Partie 4: Entity Framework Core (Suite)

(Correction)

Partie 4.1 : Configurations Fluent API

Les annotations de données sont utilisées pour configurer les classes de domaine afin de remplacer les conventions par défaut. Bien que les annotations soient simples à lire et à comprendre, elles manquent certaines fonctionnalités. L'API Fluent est un peu plus complexe à utiliser, mais fournit un ensemble de fonctionnalités beaucoup plus avancées.

Vous pouvez redéfinir la méthode OnModelCreating dans la classe contexte et utiliser ModelBuilder API pour configurer votre modèle. Il s'agit de la méthode de configuration la plus puissante, qui permet de spécifier une configuration sans modifier les classes d'entité. Dotée du niveau de priorité le plus élevé, les configurations Fluent API remplacent les conventions et les annotations de données.

Pour réduire la taille de la méthode OnModelCreating, toute les configurations d'un type d'entité peuvent être extraites dans une classe distincte qui implémente IEntityTypeConfiguration<TEntity>. Il suffit après d'appeler la méthode Configure à partir de OnModelCreating .

1. Ajouter un nouveau dossier "Configurations" dans le projet "PS.Data"
2. Ajouter la class "CategoryConfiguration" dans le dossier "Configurations":
    - Le nom de la table correspondante à l'entité catégorie dans la base de données doit être "MyCategories"
    - CategoryId est la clé primaire de la table
    - La propriété Name est obligatoire et a une longueur maximale de 50

La classe "CategoryConfiguration" imlemente l'interface "IEntityTypeConfiguration", définie dans "Microsoft.EntityFrameworkCore.Metadata.Builders", puis on passe l'entité qu'on veut configurer <Category>.

Selon la convention par défaut, EF Core nomme la table générée dans la base de données avec le même nom que la propriété DbSet qui expose l'entité. S'il n'existe aucun DbSet pour l'entité donnée, le nom de la classe est utilisé. Dans ce cas, la table sera nommée Catégories. Afin de changer le nom de cette table, nous utilisons la méthode "ToTable".
Nous utilisons la méthode "HasKey" pour configurer la clé primaire et la méthode "Property" pour appliquer des règles sur la propriété Name de la classe Category.

```
namespace PS.Data.Configurations
{
    public class CategoryConfiguration : IEntityTypeConfiguration<Category>
    {

public void Configure(EntityTypeBuilder<Category> builder)
        {
            builder.ToTable("MyCategories");
            builder.HasKey(c => c.CategoryId);
            builder.Property(c => c.Name).HasMaxLength(50).IsRequired();
        }
    }
}
```

3. Ajouter la class “ProductConfiguration” dans le dossier “Configurations”:

    ■ Configurer la relation many-to-many entre products et providers

Selon la convention par défaut, Entity Framework pour chaque relation many-to-many crée une table d'association suivant ce modèle:

> Le nom de la table sera le nom de la première entité concaténé avec le nom de la seconde entité: **ProviderProduct**

Cette table contiendra deux colonnes, les noms de ces colonnes suivront le modèle suivant: [The_name_of_the_navigation_propriety]_[The_name_of_the_primary_Key]

- **Providers_Id**
- **Products_ProductId**

Nous utilisons la méthode “HasMany” pour configurer cette relation, la méthode “ToTable” pour renommer la table d'association.

    ■ Configurer la relation one-to-many entre la class Product et Category

Nous utilisons la méthode “HasOne” pour configurer cette relation. Dans cette méthode nous transmettons la propriété de navigation **Category** de la classe Product. La méthode “WithMany” spécifie qu’une catégorie contient plusieurs produits. Dans cette méthode nous transmettons la propriété de navigation **Products** de la classe Category. Nous souhaitons également désactiver la fonction “CascadeOnDelete” pour conserver les produits après la suppression d'une catégorie associée. Pour cela, la clé étrangère “CategoryId” doit être Nullable:

```
Public virtual int? CategoryId { get; set; }
```

Lorsqu'une catégorie est supprimée, la valeur de la clé étrangère dans les produits associées est définie sur Null. “ClientSetNull” est la valeur par défaut pour les relations facultatives. Autrement dit, pour les relations qui ont des clés étrangères Nullable.

```
namespace PS.Data.Configurations
{
    public class ProductConfiguration : IEntityTypeConfiguration<Product>
    {

        public void Configure(EntityTypeBuilder<Product> builder)
        {
            //Many to Many
            builder.HasMany(p => p.Providers)
            .WithMany(v => v.Products)
            .UsingEntity(
                j => j.ToTable("Providings"));//Table d'association

            //One To Many
            builder.HasOne(p => p.MyCategory)
                .WithMany(c => c.Products)
                .HasForeignKey(p => p.CategoryId)
                .OnDelete(DeleteBehavior.ClientSetNull); //The values of foreign
key properties in dependent entities are set to null when the related principal is
deleted
        }
    }
}
```

4. Ajouter la classe “ChemicalConfiguration” dans le dossier “Configurations”:
   - Configurer le type d’entité détenu Address
   - La propriété StreeAddress a une longueur maximale de 50 et le nom de la colonne correspondante à cette propriété dans la base de données doit être “MyAddress”
   - La propriété City est obligatoire et le nom de la colonne correspondante à cette propriété dans la base de données doit être “MyCity”

```
namespace PS.Data.Configurations
{
    public class ChemicalConfiguration : IEntityTypeConfiguration<Chemical>
    {

        public void Configure(EntityTypeBuilder<Chemical> builder)
        {
            builder.OwnsOne(c => c.MyAddress, myadd =>
            {
                myadd.Property(a =>
a.StreetAddress).HasColumnName("MyStreet").HasMaxLength(50);
                myadd.Property(a => a.City).HasColumnName("MyCity").IsRequired();
            });
        }
    }
}
```

5. Mettre à jour le Context pour faire appel aux classes de configuration que nous venons de créer

■ Dans la classe PSContext on redefinit la méthode “OnModelCreating”

6. Dans la classe PSContext, configurer toute les propriétés de type string et dont le nom commence par “Name”

   ■ Le nom des colonnes correspondantes à ces propriétés dans la base de données doit être “MyName”

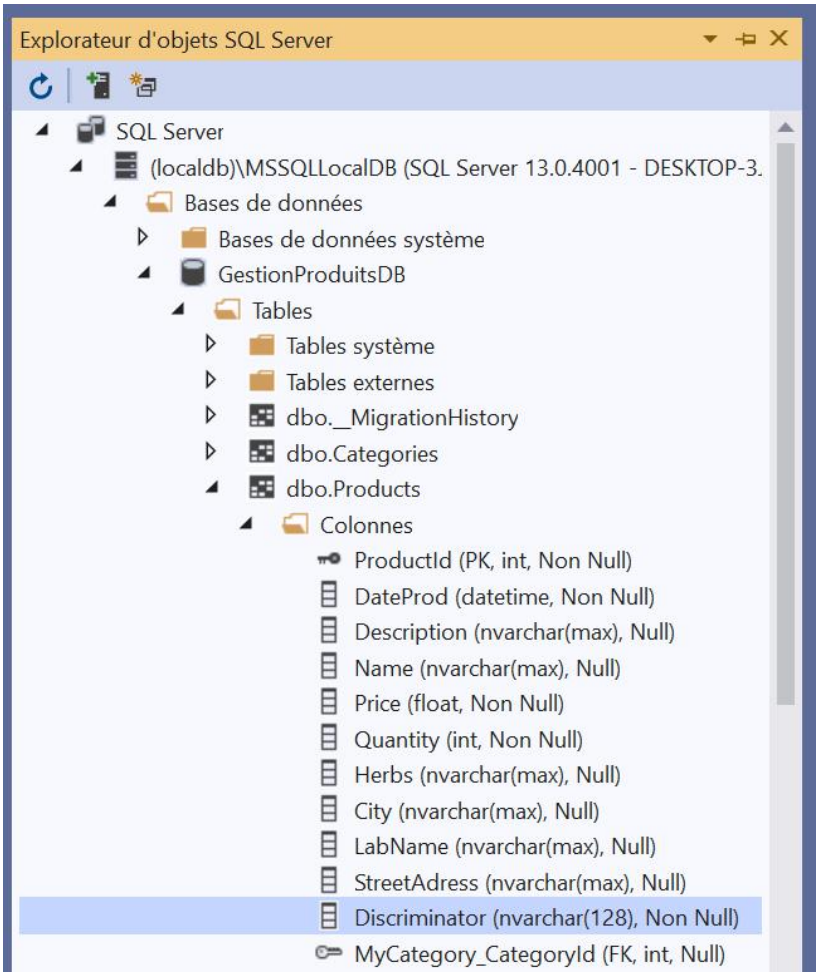
```
protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
 //Faire appel aux classes de configuration que nous venons de créer
            new CategoryConfiguration().Configure(modelBuilder.Entity<Category>());
            new ProductConfiguration().Configure(modelBuilder.Entity<Product>());
            new ChemicalConfiguration().Configure(modelBuilder.Entity<Chemical>());
//Configurer toute les propriétés de type string et dont le nom commence par “Name”
            foreach (var property in modelBuilder.Model.GetEntityTypes()
                        .SelectMany(t => t.GetProperties())
                        .Where(p => p.ClrType == typeof(string) &&
p.Name.StartsWith("Name")))
                    {
                            property.SetColumnName("MyName");
                    }
  }
```

7. Mettre à jour la base de données en utilisant la migration

Partie 4.2 : Stratégies d’heritage

Dans EF Core CodeFirst, vous pouvez faire le mapping de la stratégie d'héritage aux tables de base de données simples ou multiples en fonction de vos exigences. Voici deux approches différentes pour représenter l’héritage dans le EF Core code-First:

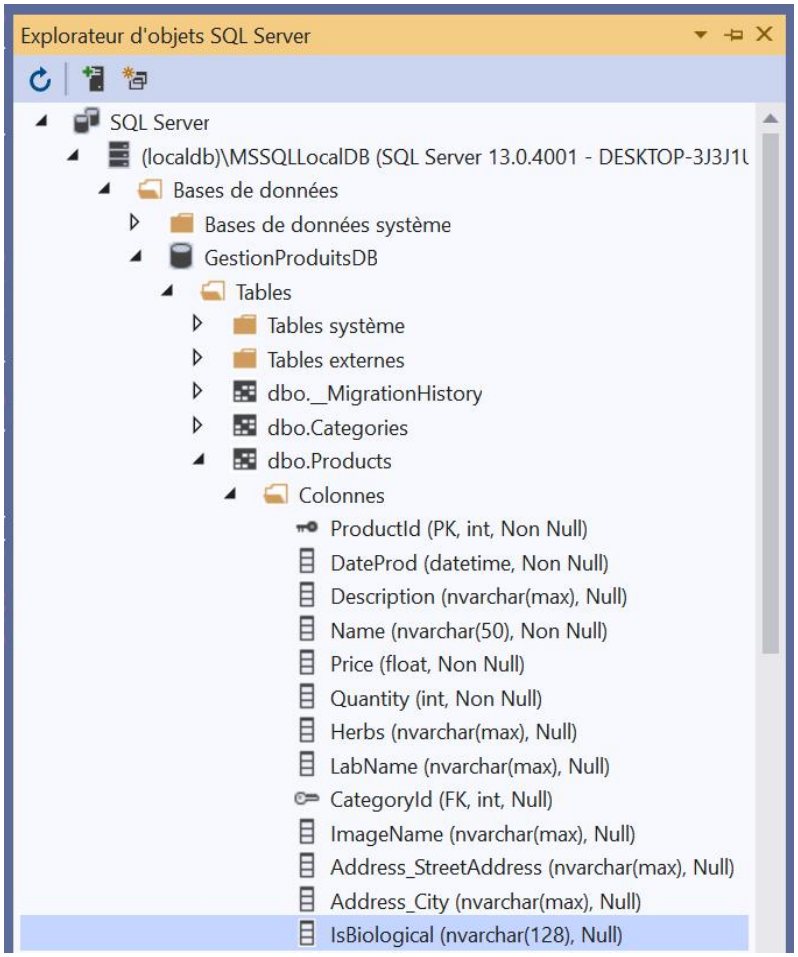
- Table-per-Hierarchy (TPH): Cette approche suggère une table pour l'ensemble de l’hiérarchie d’héritage de classe. Cette table comprend la colonne **discriminator** qui distingue entre les classes d'héritage. **Par défaut**, EF Core utilise l’approche TPH, si vous n’avez pas défini les détails de mapping pour votre hiérarchie d'héritage:



➢ Table-per-Type (TPT): Cette approche suggère une table séparée pour chaque classe.

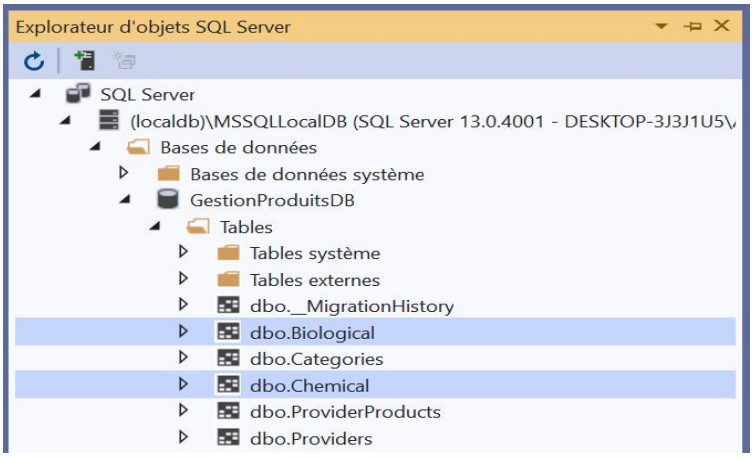
8. Configurer l'héritage schématisé dans le diagramme de classe de façon à ce que les entités soient mappées sur une seule table Products avec la colonne « IsBiological » qui prend la valeur 1 si le type de produit est Biological, la valeur 2 si le type de produit est Chemical et la valeur 0 sinon.

```
//TPH strategie d'heritage (dans la classe GPContext)
protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {

            //TPH
            modelBuilder.Entity<Product>()
                        .HasDiscriminator<int>("IsBiological")
                        .HasValue<Biological>(1)
                        .HasValue<Chemical>(2)
                        .HasValue<Product>(0);

        }
```



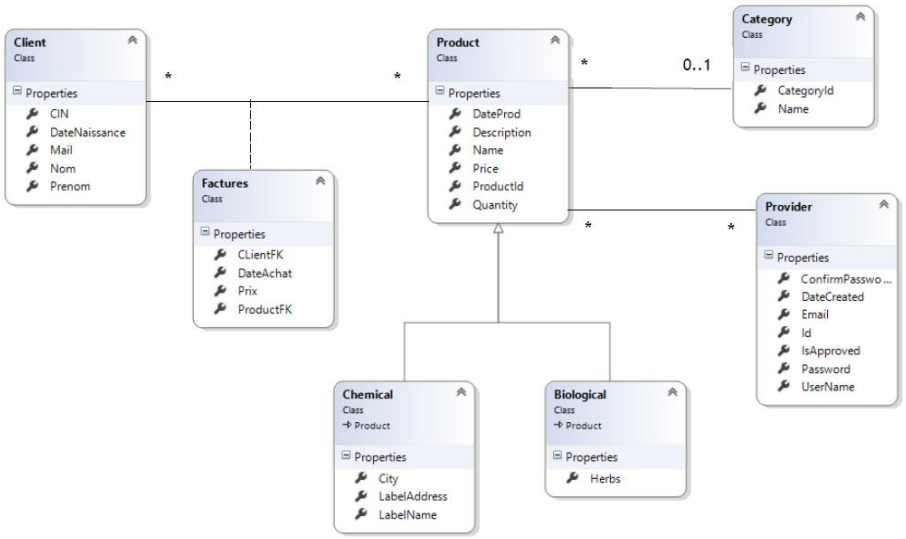
9. Configurer l'héritage schématisé dans le diagramme de classe à ce que les entités soient mappées sur 3 tables.

```
//TPT strategie d'heritage (dans la classe GPContext)
            modelBuilder.Entity<Biological>().ToTable("Biologicals");
            modelBuilder.Entity<Chemical>().ToTable("Chemicals");
```



Partie 4.3 : Table porteuse de données

10. Dans le projet “PS.Domain” ajouter la classe Client et la classe Facture schématisées dans le diagramme suivant :




```
 namespace PS.Domain
{
   public class Facture
     {
         public DateTime DateAchat { get; set; }
         public int ProductFk { get; set; }
         public int ClientFk { get; set; }
         public int Prix { get; set; }
         public Client Client { get; set; }
         public Product Product { get; set; }
     }
}
```

```
namespace PS.Domain
{
    public class Client
    {[Key]
        public int CIN { get; set; }
        public string Nom { get; set; }
        public string Prenom { get; set; }
        public DateTime DateNaissance { get; set; }
        public string Mail { get; set; }
        //prop de navigation
        public IList<Product> Products { get; set; }
        public IList<Facture> Factures { get; set; }
    }
}
```

N’oublier pas d’ajouter la propriété de navigation dans la classe Product:

```
Public IList<Facture> Factures { get; set; }
```

11. Configurer la classe Facture

On peut configurer la classe Facture soit dans la classe `ProductConfiguration` en décomposant la relation many to many entre Product et Client en deux relations one to many

```
public class ProductConfiguration : IEntityTypeConfiguration<Product>
    {
        public void Configure(EntityTypeBuilder<Product> builder)
        {
            //Many to Many
            builder.HasMany(p => p.Clients)
            .WithMany(c => c.Products)
            .UsingEntity<Facture>(
                j => j
                    .HasOne(f => f.Client)
                    .WithMany(c => c.Factures)
                    .HasForeignKey(c => c.ClientFk),
                j => j
                    .HasOne(f => f.Product)
                    .WithMany(p => p.Factures)
                    .HasForeignKey(p => p.ProductFk),
                j =>
                {
                    j.Property(f =>
f.DateAchat).HasDefaultValueSql("CURRENT_TIMESTAMP");
                    j.HasKey(f => new { f.DateAchat, f.ClientFk, f.ProductFk });
                });
        }
    }
```

ou bien en créant une nouvelle classe de configuration `FactureConfiguration:`

```
public class FactureConfiguration : IEntityTypeConfiguration<Facture>
    {
        public void Configure(EntityTypeBuilder<Facture> builder)
        {
            builder.HasKey(f => new
            {
                f.DateAchat,
                f.ClientFk,
                f.ProductFk
            });

            builder.HasOne(f => f.Client)
            .WithMany(c => c.Factures)
            .HasForeignKey(f => f.ClientFk);

            builder.HasOne(f => f.Product)
           .WithMany(p => p.Factures)
           .HasForeignKey(f => f.ProductFk);
        }
    }
}
```

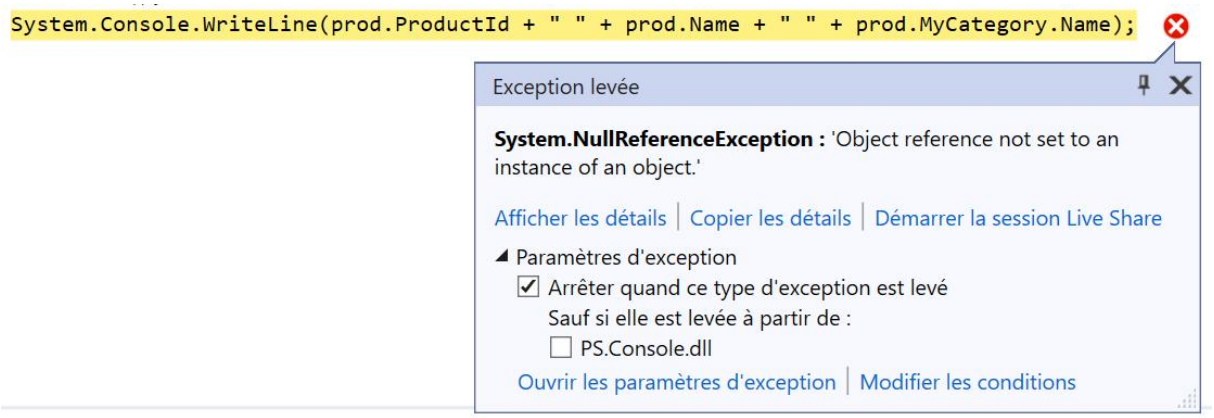
12. Mettre à jour la base de données en utilisant la migration.

Partie 4.4 : Chargement des données associées

14. Faite une opération d'ajout d'un produit en lui attribuant une catégorie et une opération d'affichage de ce produit puis lancer l'application.

```
namespace PS.Console
{
    class Program
    {
        static void Main(string[] args)


        {
            using (var context = new PSContext())
            {
                //Create
                System.Console.WriteLine("Create");
                //Instancier un objet Category
                Category C = new Category { Name = "Cat1" };
                //Instancier un objet Product
                Product P = new Product { Name = "Prod1", DateProd = DateTime.Now,
                                          MyCat = C };
                //Ajouter l'objet au DBSET
                context.Products.Add(P);
                //Persister les données
                context.SaveChanges();


                //Read All
                System.Console.WriteLine("Read All");
                foreach (Product p in context.Products)
                {
                    System.Console.WriteLine(p.ProductId + " " + p.Name + " " +
p.MyCat.Name);
                }
            }
        }
       }
}
```

15. Commenter la partie d'ajout des données et relancer l'application. L'application nous renvoie cette exception:



La categorie est ajoutée dans la base de données, mais une exception est levée (Category=Null).

Par défaut, EF Core ne charge pas les données associées. La solution est d'utiliser le chargement différé (Lazy Loading).

Le chargement différé signifie que les données associées sont chargées de façon transparente à partir de la base de données lors de l'accès à la propriété de navigation.

La façon la plus simple d'utiliser le chargement différé est d'installer le package Microsoft.EntityFrameworkCore.Proxies et de l'activer avec un appel à UseLazyLoadingProxies

16. Installer EntityFrameworkCore.Proxies dans le projet PS.Data.

17. Activer le LazyLoading dans la méthode OnConfiguring de la classe PSContext.

```
 protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseLazyLoadingProxies();
            optionsBuilder.UseSqlServer(@"Data
Source=(localdb)\mssqllocaldb;Initial Catalog=ProductStoreDB;Integrated
Security=true");
            base.OnConfiguring(optionsBuilder);
        }
```

18. Décorer toutes les propriétés de navigation des classes d'entités avec le mot clé virtual.

EF Core active le chargement différé pour n'importe quelle propriété de navigation qui peut être substituée, c'est-à-dire qui doit être virtual et sur une classe qui peut être héritée.

19. Relancer l'application