

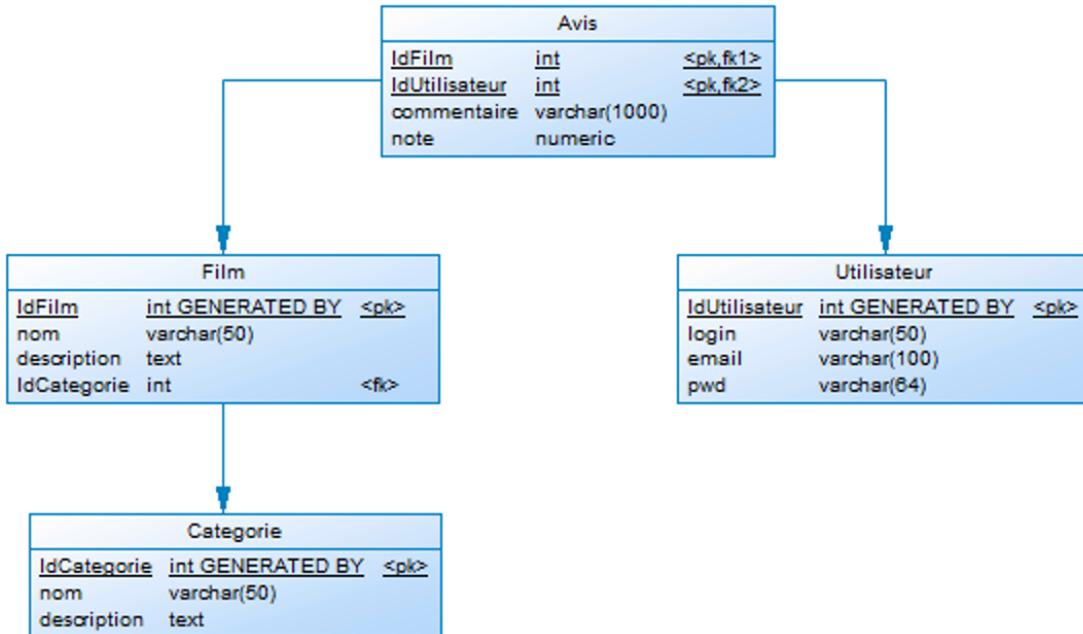
1. Mise en place et bases du fonctionnement

1.1. Mise en place

- a) Création de la base de données sous PostgreSQL.

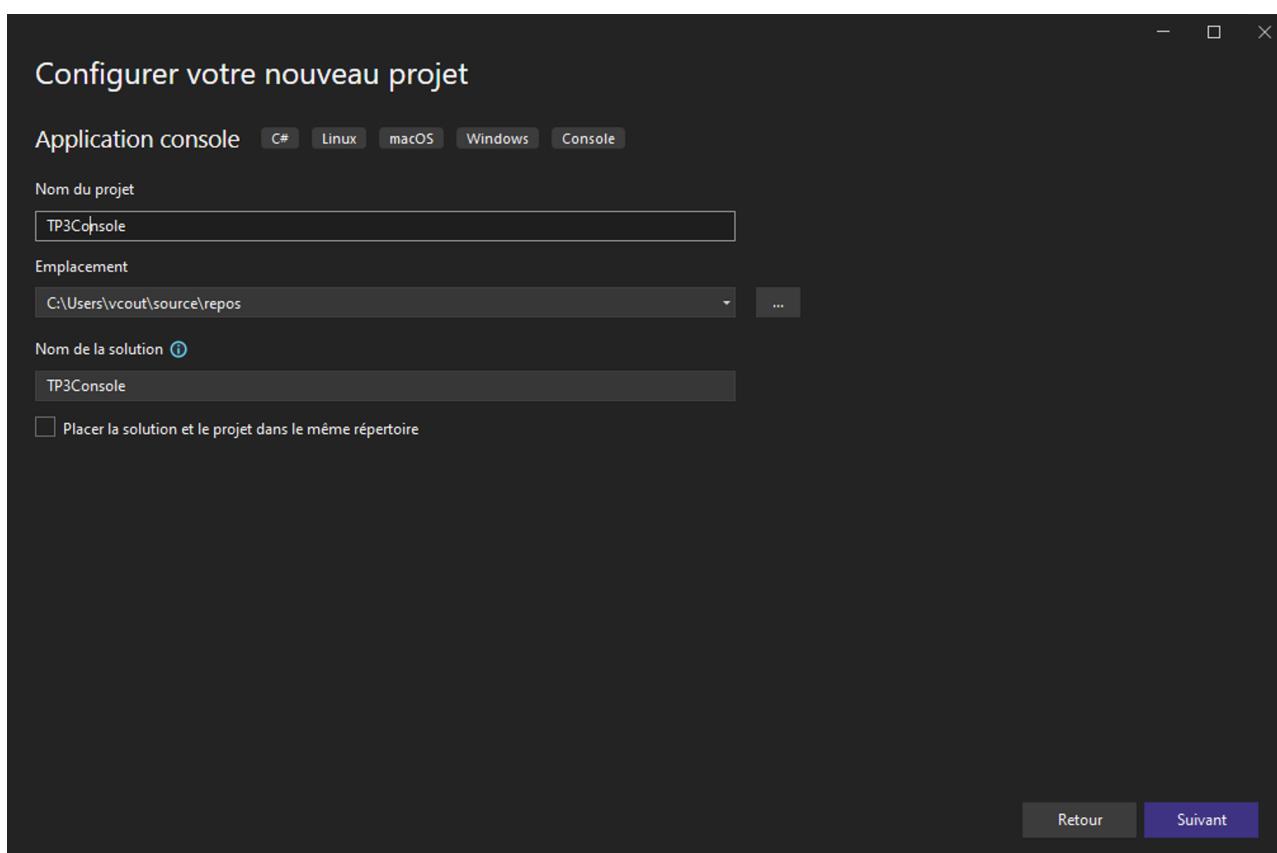
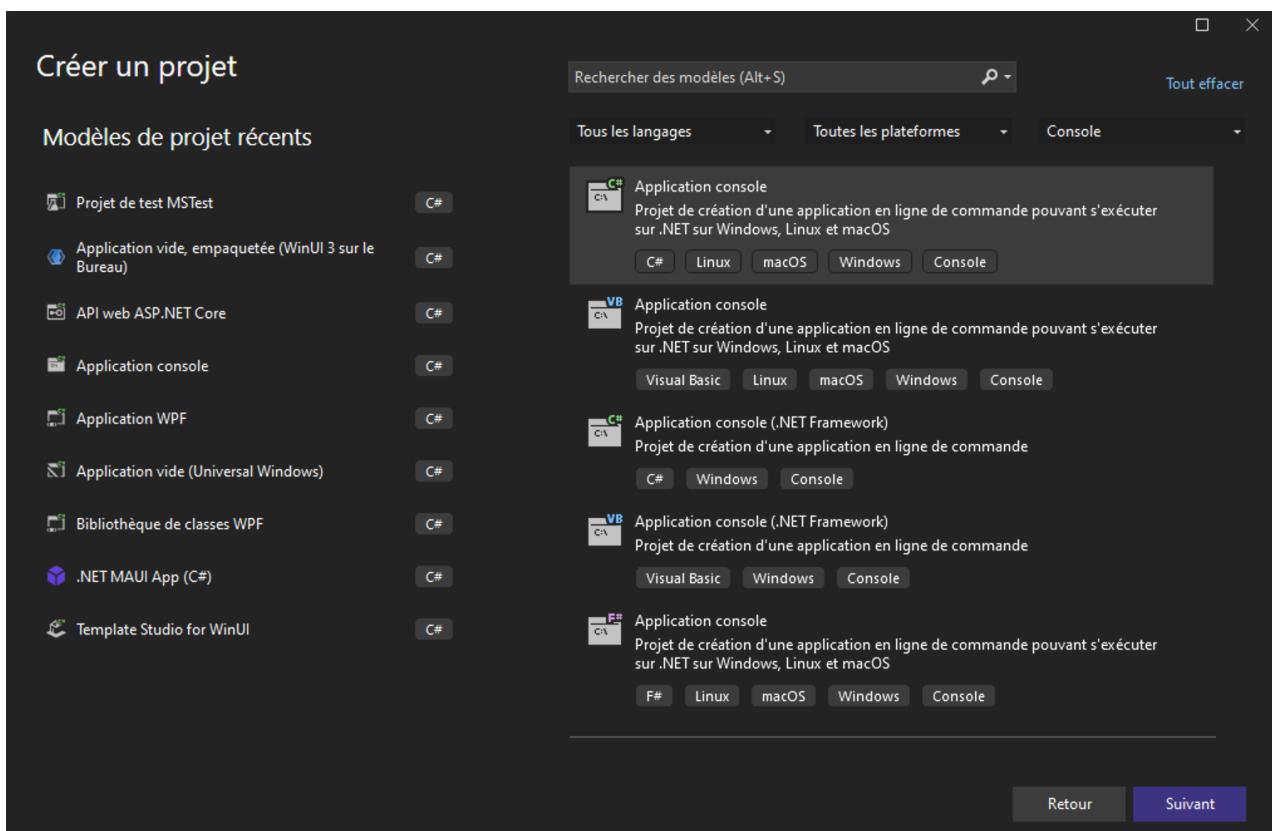
En local, sous PgAdmin 4, créer une base FilmsDB puis exécuter le script `scriptFilmsPostgreSQL.sql`.

Schéma de la base de données :

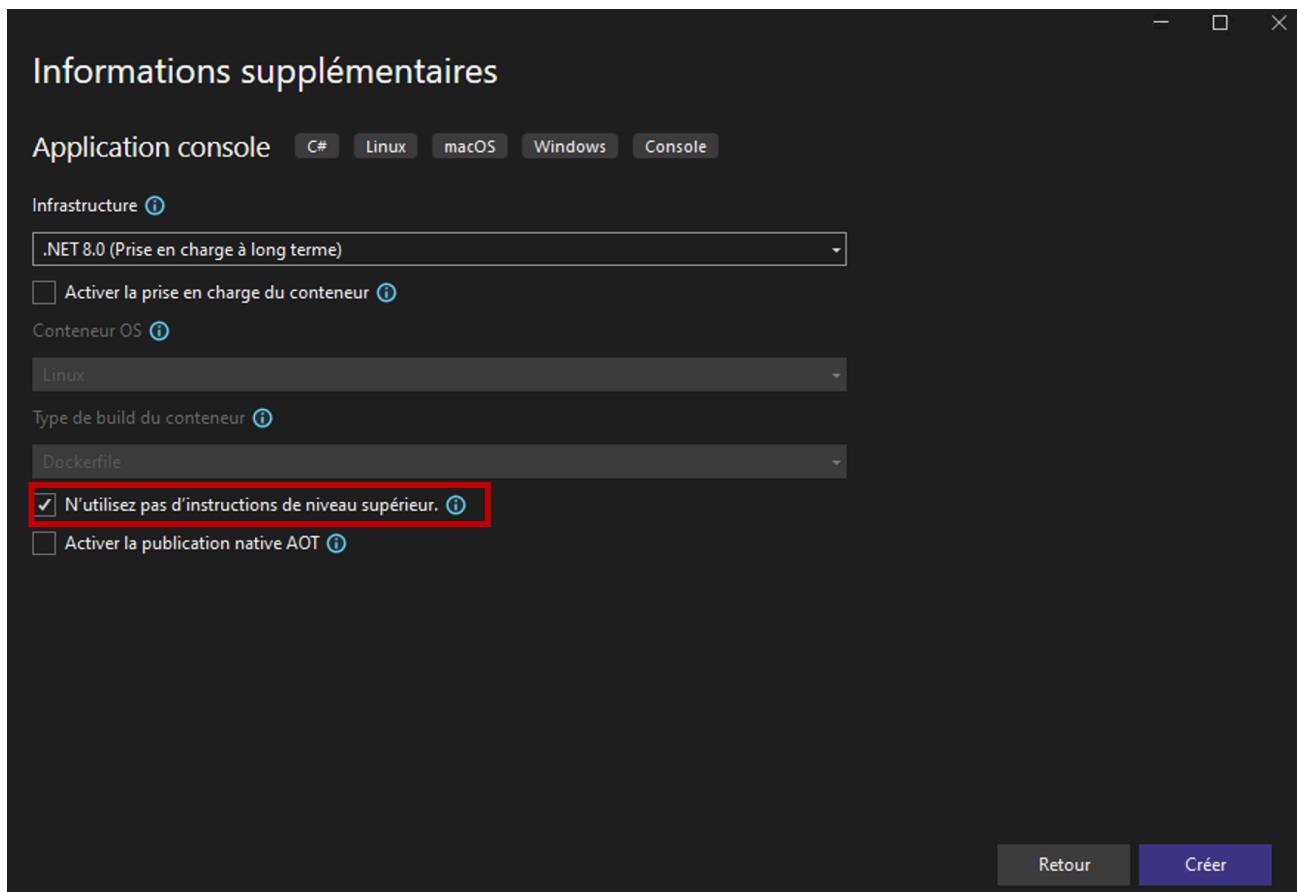


- b) Création du projet .NET

Pour simplifier, nous allons travailler sur une application console. Lancer Visual Studio et créer une nouvelle *Application console* nommée **TP3console**.



Bien cocher « N'utilisez pas d'instruction de niveau supérieur » pour avoir accès à la méthode Main.



c) Création du modèle de données

Installer les packages NuGet suivants :

- Npgsql.EntityFrameworkCore.PostgreSQL : Nom complet du fournisseur de données (provider). **Npgsql est le driver PostgreSQL.**
- Npgsql.EntityFrameworkCore.PostgreSQL.Design : Les librairies Design contiennent les instructions utilisées par le fournisseur de données pour générer le code source .NET à partir de la structure de la base de données (scaffolding).
- Microsoft.EntityFrameworkCore.Tools : Contient les commandes PowerShell de scaffolding.

ATTENTION à installer les packages Microsoft.EntityFrameworkCore.Tools et Npgsql.EntityFrameworkCore.PostgreSQL dans une version compatible avec votre version de .NET ET dans la même version (par exemple, la version 8.0.11 sur .NET 8). Utiliser TOUJOURS les mêmes versions pour ces 2 packages.

Comme dans le TP2, nous allons créer les classes de modèle en utilisant l'API Fluent puis les data annotations.

Création du modèle en utilisant l'API Fluent

Dans la console du gestionnaire de package exécuter la commande suivante :

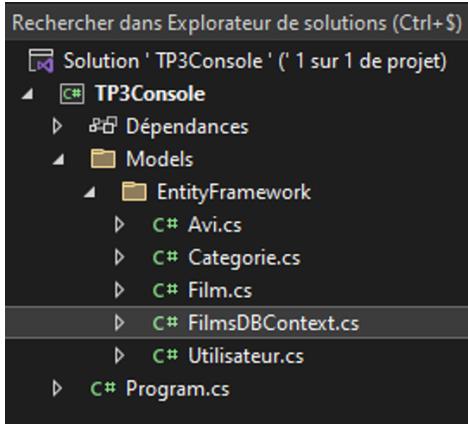
```
Scaffold-DbContext "Server=localhost;port=5432;Database=FilmsDB; uid=postgres;
password=postgres;" -Provider Npgsql.EntityFrameworkCore.PostgreSQL -OutputDir
Models/EntityFramework
```

```
Console du Gestionnaire de package
Source de packages : Tout   Projet par défaut : TP2Console
Chaque package vous est concédé sous licence par son propriétaire. NuGet n'est pas responsable des packages tiers et n'octroie aucune licence les concernant. Certains packages peuvent inclure des dépendances régies par des licences supplémentaires. Suivez l'URL (flux) de la source de packages pour déterminer les dépendances éventuelles.

Version de l'hôte de la Console du Gestionnaire de package 5.10.0.7240
Tapez 'get-help NUGet' pour afficher toutes les commandes NUGet disponibles.

PM> Scaffold-DbContext "Server=localhost;port=5432;Database=FilmsDB; uid=postgres;
password=postgres;" -Provider Npgsql.EntityFrameworkCore.PostgreSQL -OutputDir Models/EntityFramework
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string, you should move it out of source code. You can avoid scaffolding the connection string by using the Name= syntax to
read it from configuration - see https://go.microsoft.com/fwlink/?LinkId=2131148. For more guidance on storing connection strings, see http://go.microsoft.com/fwlink/?LinkId=723263.
PM>
```

Solution :



Contenu du fichier Avi.cs :

```
5
6     <!-- public partial class Avi
7     {
8         public int Idfilm { get; set; }
9
10        public int Idutilisateur { get; set; }
11
12        public string? Commentaire { get; set; }
13
14        public decimal Note { get; set; }
15
16        public virtual Film IdfilmNavigation { get; set; } = null!;
17
18        public virtual Utilisateur IdutilisateurNavigation { get; set; } = null!;
19    }
20
```

On remarque 2 propriétés de navigation, créées à partir des clés étrangères, permettant d'accéder aux objets associés. Ces propriétés sont toujours `virtual` :

<https://learn.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/virtual>

Les autres propriétés sont des propriétés standards qui sont le reflet de la base de données.

`Partial` signifie qu'il s'agit d'une classe partielle dont le code pourra être complété par une ou plusieurs autres classes partielles ayant le même nom (de classe).

Contenu du fichier Film.cs :

```
5
6     <!-- public partial class Film
7     {
8         public int Idfilm { get; set; }
9
10        public string Nom { get; set; } = null!;
11
12        public string? Description { get; set; }
13
14        public int Idcategorie { get; set; }
15
16        public virtual ICollection<Avi> Avis { get; set; } = new List<Avi>();
17
18        public virtual Categorie IdcategorieNavigation { get; set; } = null!;
19    }
20
```

On remarque la `ICollection<Avi>` qui permet de récupérer la collection d'avis liée à un film. Il s'agit aussi d'une propriété de navigation.

`=null!` permet d'obliger la saisie du nom (\Leftrightarrow NOT NULL dans la base de données).

Contenu du fichier FilmsDbContext.cs :

- La classe `FilmsDbContext` permet de se connecter à la base de données et va s'occuper des opérations de création, lecture, mise à jour et suppression pour nous (CRUD). Elle hérite de `DbContext` :
<https://learn.microsoft.com/fr-fr/ef/core/api/microsoft.entityframeworkcore.dbcontext>
- On remarque la property `Avis` qui permet de manipuler les objets métiers (avis) et notamment de les récupérer (`get`) sous la forme d'une collection (`DbSet`). Idem, pour les 3 autres properties. La

classe DbSet représente une collection de toutes les entités dans le contexte. Une entité représente une table ou une vue SQL.

```
public virtual DbSet<Avi> Avis { get; set; }

public virtual DbSet<Categorie> Categories { get; set; }

public virtual DbSet<Film> Films { get; set; }

public virtual DbSet<Utilisateur> Utilisateurs { get; set; }
```

- Pour se connecter à la base de données, il faut implémenter la méthode OnConfiguring.
`protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
 => optionsBuilder.UseNpgsql("Server=localhost;port=5432;Database=FilmsDB; uid=postgres;
password=postgres;");`
On y retrouve la chaîne de connexion saisie lors du scaffolding.
- Pour créer le modèle, il faut implémenter la méthode OnModelCreating de la classe héritée de DbContext. Exemple de contenu :
 - o Définition du mapping entre la base de données et la classe Film :

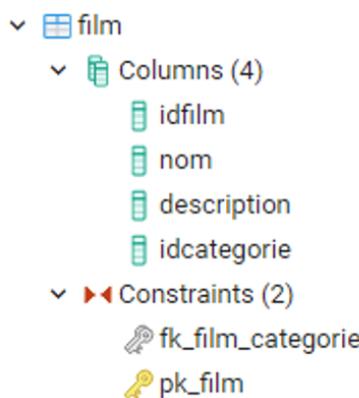
```
modelBuilder.Entity<Film>(entity =>
{
    entity.HasKey(e => e.Idfilm).HasName("pk_film");

    entity.ToTable("film");

    entity.Property(e => e.Idfilm).HasColumnName("idfilm");
    entity.Property(e => e.Description).HasColumnName("description");
    entity.Property(e => e.Idcategorie).HasColumnName("idcategorie");
    entity.Property(e => e.Nom)
        .HasMaxLength(50)
        .HasColumnName("nom");

    entity.HasOne(d => d.IdcategorieNavigation).WithMany(p => p.Films)
        .HasForeignKey(d => d.Idcategorie)
        .onDelete(DeleteBehavior.Restrict)
        .HasConstraintName("fk_film_categorie");
});
```

Rappel : définition de la table :



`HasKey` permet de définir la clé primaire et `HasName` son nom.

La propriété `Idfilm` (`e.Idfilm`) de la classe correspond à la colonne `idfilm` de la table. Il n'y a pas d'autres caractéristiques définies ici car les types correspondent : `int` (de la propriété) `<->` `int` ou `integer` (type de colonne de la table).

Pour la propriété `Description`, le type de la colonne n'est pas précisé car par défaut `string` correspond au type SQL `varchar`. Pourtant le type SQL utilisé dans le script est `text`, mais dans PostgreSQL, `text` est remplacé par `varchar` (`text` est un synonyme de `varchar`).

Pour la propriété `Nom`, on remarque que le nombre de caractères maximum de la chaîne est précisé (`HasMaxLength`).

- o Définition du mapping entre la base de données et la classe Avi :

```

modelBuilder.Entity<Avi>(entity =>
{
    entity.HasKey(e => new { e.Idfilm, e.Idutilisateur }).HasName("pk_avis");

    entity.ToTable("avis");

    entity.Property(e => e.Idfilm).HasColumnName("idfilm");
    entity.Property(e => e.Idutilisateur).HasColumnName("idutilisateur");
    entity.Property(e => e.Commentaire)
        .HasMaxLength(1000)
        .HasColumnName("commentaire");
    entity.Property(e => e.Note).HasColumnName("note");

    entity.HasOne(d => d.IdfilmNavigation).WithMany(p => p.Avis)
        .HasForeignKey(d => d.Idfilm)
        .onDelete(DeleteBehavior.Restrict)
        .HasConstraintName("fk_avis_film");

    entity.HasOne(d => d.IdutilisateurNavigation).WithMany(p => p.Avis)
        .HasForeignKey(d => d.Idutilisateur)
        .onDelete(DeleteBehavior.Restrict)
        .HasConstraintName("fk_avis_utilisateur");
});

```

Rappel : définition de la table :



```

entity.HasOne(d => d.IdfilmNavigation).WithMany(p => p.Avis)
    .HasForeignKey(d => d.Idfilm)
    .onDelete(DeleteBehavior.Restrict)
    .HasConstraintName("fk_avis_film");

```

HasOne () .WithMany () .HasForeignKey () ... permet de définir une clé étrangère, ici entre la tables Avis (FK) et Film (PK). On y retrouve le nom de la property de navigation dans la classe Avi (IdfilmNavigation) (HasOne ici car un avis est toujours lié à un seul film, HasMany sinon), le nom de la property de navigation associée dans la classe Film (Avis) (WithMany car à un film correspond plusieurs avis, WithOne sinon), le nom de la property qui est FK dans la classe Avi (IdFilm), le type de suppression des enregistrements liés .onDelete(DeleteBehavior.Restrict) (mode de suppression des avis quand un film sera supprimé) et le nom de la FK dans la table (HasConstraintName ("fk_avis_film")). Les modes de suppression possibles sont Cascade, ClientSetNull, SetNull ou Restrict. Le comportement de chaque mode est spécifique au type de clé étrangère. En effet, il existe deux types de clé étrangère :

- Clé étrangère facultative (elle peut être de valeur null).
- Clé étrangère obligatoire.

Table 1. Clé étrangère facultative

Nom du comportement	Effet sur les entités dépendantes en mémoire	Effet sur les entités dépendantes en base de données
Cascade	Suppression	Suppression
ClientSetNull (Par défaut)	Valeur à NULL	Aucun
SetNull	Valeur à NULL	Valeur à NULL
Restrict	Aucun	Aucun

Table 2. Clé étrangère obligatoire

Nom du comportement	Effet sur les entités dépendantes en mémoire	Effet sur les entités dépendantes en base de données
Cascade (Par défaut)	Suppression	Suppression
ClientSetNull	Lève une exception	Aucun
SetNull	Lève une exception	Lève une exception
Restrict	Aucun	Aucun

Le code que nous venons d'analyser correspond à l'API Fluent :

<https://www.learnentityframeworkcore.com/configuration/fluent-api>

Création du modèle en utilisant les Data Annotations

Exécuter la commande suivante dans la console du gestionnaire de package NuGet :

```
Scaffold-DbContext "Server=localhost;port=5432;Database=FilmsDB; uid=postgres; password=postgres;" -Provider Npgsql.EntityFrameworkCore.PostgreSQL -OutputDir Models/EntityFramework -DataAnnotations -force
```

Classe Avi générée :

```
[PrimaryKey("Idfilm", "Idutilisateur")]
[Table("avis")]
public partial class Avi
{
    [Key]
    [Column("idfilm")]
    public int Idfilm { get; set; }

    [Key]
    [Column("idutilisateur")]
    public int Idutilisateur { get; set; }

    [Column("commentaire")]
    [StringLength(1000)]
    public string? Commentaire { get; set; }

    [Column("note")]
    public decimal Note { get; set; }

    [ForeignKey("Idfilm")]
    [InverseProperty("Avis")]
    public virtual Film IdfilmNavigation { get; set; } = null!;

    [ForeignKey("Idutilisateur")]
    [InverseProperty("Avis")]
    public virtual Utilisateur IdutilisateurNavigation { get; set; } = null!;
}
```

Plutôt que de mettre en dur les noms des properties auxquelles font référence les annotations [ForeignKey] et [InverseProperty], il est préférable d'utiliser nameof.

```

[PrimaryKey("Idfilm", "Idutilisateur")]
[Table("avis")]
public partial class Avi
{
    [Key]
    [Column("idfilm")]
    public int Idfilm { get; set; }

    [Key]
    [Column("idutilisateur")]
    public int Idutilisateur { get; set; }

    [Column("commentaire")]
    [StringLength(1000)]
    public string? Commentaire { get; set; }

    [Column("note")]
    public decimal Note { get; set; }

    [ForeignKey(nameof(Idfilm))]
    [InverseProperty(nameof(Film.Avis))]
    public virtual Film IdfilmNavigation { get; set; } = null!;

    [ForeignKey(nameof(Idutilisateur))]
    [InverseProperty(nameof(Utilisateur.Avis))]
    public virtual Utilisateur IdutilisateurNavigation { get; set; } = null!;
}

```

Principales annotations :

- `[Table]` : Permet de définir le nom de la table dans la base de données, sinon le nom de la table sera le même que celui de la classe. On peut aussi spécifier un schéma spécifique. Correspond à `toTable` de l'API Fluent.
- `[Column]` : Permet de spécifier le nom de la colonne. Correspond à `HasName` de l'API Fluent. On peut aussi spécifier le type ainsi que l'ordre.
- `[Key]` : Permet de définir la clé primaire. On peut retrouver cette annotation plusieurs fois dans le cas d'une clé primaire composée (dans ce cas, il est possible d'ajouter une annotation `[Column(Order)]`, par ex. `[Column(Order=1)]`). Correspond à `HasKey` de l'API Fluent. <http://www.entityframeworktutorial.net/code-first/key-dataannotations-attribute-in-code-first.aspx>
- `[PrimaryKey]` permet de définir une PK portant sur plusieurs champs.
- `[ForeignKey(nameof(MyProperty))]` : Permet de spécifier le nom de la property qui est clé étrangère dans la classe actuelle. `ForeignKey` est toujours associée à l'instruction `InverseProperty` qui permet d'indiquer la property liée dans la classe associée.
- L'annotation `[InverseProperty]` permet de définir la propriété de navigation « inverse » dans la classe liée.

Exemple dans `Avi` :

```

[ForeignKey(nameof(Idfilm))]
[InverseProperty(nameof(Film.Avis))]
public virtual Film IdfilmNavigation { get; set; } = null!;

```

A comparer dans `Film` à :

```

[InverseProperty(nameof(Avi.IdfilmNavigation))]
public virtual ICollection<Avi> Avis { get; set; } = new List<Avi>();

```

Plus d'explications ici :

<https://stackoverflow.com/questions/40480601/what-is-difference-between-inverse-property-and-foreign-key-in-entity-framework>

Classe `Film` (après modification avec `nameof`) :

```

[Table("film")]
public partial class Film
{
    [Key]
    [Column("idfilm")]
    public int Idfilm { get; set; }

    [Column("nom")]
    [StringLength(50)]
    public string Nom { get; set; } = null!;

    [Column("description")]
    public string? Description { get; set; }

    [Column("idcategorie")]
    public int Idcategorie { get; set; }

    [InverseProperty(nameof(Avi.IdfilmNavigation))]
    public virtual ICollection<Avi> Avis { get; set; } = new List<Avi>();

    [ForeignKey("Idcategorie")]
    [InverseProperty(nameof(Categorie.Films))]
    public virtual Categorie IdcategorieNavigation { get; set; } = null!;
}

```

Autres annotations possibles :

- `[DataBaseGenerated]` : Permet de spécifier comment les valeurs seront générées par le SGBD :
 - `None` : Elle ne sera pas générée automatiquement par le serveur de BDD. L'utilisateur devra spécifier la valeur de la clé primaire. Correspond à `ValueGeneratedNever` de l'API Fluent.
 - `Identity` : Permet de spécifier que la valeur sera générée à l'insertion d'une ligne. L'attribut ne pourra pas être modifié ultérieurement. Correspond à `ValueGeneratedOnAdd` de l'API Fluent.
 - `Computed` : Permet de spécifier que la valeur sera générée à chaque modification de la ligne. Correspond à `ValueGeneratedOnAddOrUpdate` de l'API Fluent.
- `[Required]` permet de spécifier que la valeur ne peut pas être `null` (autre possibilité : `=null!`). Dans la base de données, ce sera une valeur NOT NULL. Correspond à `IsRequired` de l'API Fluent.
- `[MaxLength]` permet de spécifier la taille maximum du champ (en caractère ou en octet). Correspond à `HasMaxLength` de l'API Fluent.
- `[StringLength]` est comparable à `MaxLength`. On peut spécifier `MinimumLength` qui n'aura aucune incidence sur la BD mais qui permettra dans les formulaires du site web de définir des règles de caractères minimums sur l'attribut. `StringLength` n'a pas d'équivalent dans l'API Fluent, utiliser `HasMaxLength`.
- `[NotMapped]` permet de spécifier qu'une property ne sera pas liée à un champ de la base de données. Correspond à `ignore` dans l'API Fluent.
- Contraintes de validation :
 - `[Phone]`, `[CreditCard]`, `[EmailAddress]`, `[Range]` :
<https://docs.microsoft.com/fr-fr/dotnet/api/system.componentmodel.dataannotations.rangeattribute>
 - `[RegularExpression]` :
<https://docs.microsoft.com/fr-fr/dotnet/api/system.componentmodel.dataannotations.regularexpressionattribute>
 - `[EnumDataType]` :
<https://docs.microsoft.com/fr-fr/dotnet/api/system.componentmodel.dataannotations.enumdatatypeattribute>
 - `[Url]`, etc.

Toutes les annotations possibles ici :

<https://learn.microsoft.com/fr-fr/ef/core/modeling/relationships/mapping-attributes>

Classe de contexte :

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Avi>(entity =>
    {
        entity.HasKey(e => new { e.Idfilm, e.Idutilisateur }).HasName("pk_avis");

        entity.HasOne(d => d.IdfilmNavigation).WithMany(p => p.Avis)
            .OnDelete(DeleteBehavior.Restrict)
            .HasConstraintName("fk_avis_film");

        entity.HasOne(d => d.IdutilisateurNavigation).WithMany(p => p.Avis)
            .OnDelete(DeleteBehavior.Restrict)
            .HasConstraintName("fk_avis_utilisateur");
    });

    modelBuilder.Entity<Categorie>(entity =>
    {
        entity.HasKey(e => e.Idcategorie).HasName("pk_categorie");
    });

    modelBuilder.Entity<Film>(entity =>
    {
        entity.HasKey(e => e.Idfilm).HasName("pk_film");

        entity.HasOne(d => d.IdcategorieNavigation).WithMany(p => p.Films)
            .OnDelete(DeleteBehavior.Restrict)
            .HasConstraintName("fk_film_categorie");
    });

    modelBuilder.Entity<Utilisateur>(entity =>
    {
        entity.HasKey(e => e.Idutilisateur).HasName("pk_utilisateur");
    });
}

OnModelCreatingPartial(modelBuilder);
}

```

Le code est plus concis car une importante partie des caractéristiques portant sur les properties ont été codées sous la forme de Data annotations.

Pour chaque entité, le code n'est dédié qu'à la création de la clé primaire et des FK.

Explications de la clé étrangère suivante :

```

modelBuilder.Entity<Avi>(entity =>
{
    ...
    entity.HasOne(d => d.IdfilmNavigation).WithMany(p => p.Avis)
        .OnDelete(DeleteBehavior.Restrict)
        .HasConstraintName("fk_avis_film");
}

```

- `entity.HasOne(d => d.IdfilmNavigation)` : propriété de navigation de la classe Avi (table Avis -> Film). HasOne car un seul film lié à l'avis. On remarque `[InverseProperty(nameof(Film.Avis))]` devant la propriété de navigation `IdfilmNavigation` de la classe Avi permettant de définir la property de navigation liée dans la table Film.
- `.WithMany(p => p.Avis)` : Propriété de navigation de la classe Film (WithMany car collection de <Avi>). On remarque `[InverseProperty(nameof(Avi.IdfilmNavigation))]` devant la propriété de navigation `Avis` de la classe Avi permettant de faire la liaison entre les 2 properties de navigation. InverseProperty permet de définir une "référence croisée".
- **Il faut toujours utiliser `d` et `p` :** `d` signifie entité dépendante (celle qui contient la FK) et `p` entité principale (celle qui contient la PK). Plus de détail ici : <https://learn.microsoft.com/fr-fr/ef/core/modeling/relationships?tabs=fluent-api%2Cfluent-api-simple-key%2Csimple-key>

a) Modification du modèle

Comme nous l'avons vu en fin de TP2, les classes du modèle et le code de la classe de contexte ne doivent pas être modifiés, car les modifications seront perdues lors de la régénération du modèle.

Pour modifier le modèle, la seule possibilité est d'utiliser des classes partielles (mot clé partial). Les classes du modèle pourront ainsi être modifiées sans risque de perdre les modifications. Des classes partielles ont le même nom avec le mot clé `partial` indiquant qu'elles sont complétées par d'autres classes (`public partial class MaClasse dans les 2 cas`) ; **seul le nom du fichier doit différer**.

1.2. Utilisation basique d'Entity Framework

Entity Framework est un ORM « à état », c'est-à-dire qu'il travaille sur des objets stockés dans un *context* et la synchronisation entre ce cache et la base de données n'est pas systématique (c'est le développeur qui devra demander cette synchronisation en appelant la méthode `SaveChanges()`).

Voilà un exemple basique de modification d'une entité avec Entity Framework. Ecrire puis tester ce code.

```
using TP2Console.Models.EntityFramework;

namespace TP2Console
{
    internal class Program
    {
        static void Main(string[] args)
        {
            using (var ctx = new FilmsDbContext())
            {

                //Requête SELECT
                Film titanic = ctx.Films.First(f => f.Nom.Contains("Titanic"));

                //Modification de l'entité (dans le contexte seulement)
                titanic.Description = "Un bateau échoué. Date : " + DateTime.Now;

                //Sauvegarde du contexte => Application de la modification dans la BD
                int nbchanges = ctx.SaveChanges();

                Console.WriteLine("Nombre d'enregistrements modifiés ou ajoutés : " + nbchanges);
            }
        }
    }
}
```

Remarque : ne pas oublier d'ajouter le namespaces `System.Linq`.

Résultat dans la base de données :

	idfilm [PK] integer	nom character varying (50)	description text	idcategorie integer
1	3	Titanic	Un bateau échoué. Date : 19/09/2024 15:25:21	5

Requêtes Linq

Vous remarquerez la syntaxe Linq en ligne 14. C'est une syntaxe qui semble au départ assez peu « naturelle » mais qui permettra une fois maîtrisée de simplifier grandement la gestion des listes (car Linq ne s'applique pas qu'à EF Core, mais à n'importe quelle collection).

La syntaxe ci-dessus utilise les expressions lambda (`s => s...`). Ce sont en fait des fonctions anonymes. Une autre syntaxe, qui ressemble un peu plus au SQL existe. Par exemple, la même requête (sélection de la catégorie actions) peut s'écrire :

```
Film titanic = (from f in ctx.Films
                 where f.Nom == "Titanic"
                 select f).First();
```

Il n'y a pas d'avantage ou d'inconvénient. Ce sont juste 2 syntaxes différentes, à vous de choisir celle que vous préférez !

Linq fournit un grand nombre de fonctions qui permettent de faire presque toutes les opérations SQL. Les principales fonctions Linq utilisées sont :

- Select
- Where
- First / FirstOrDefault
- Single / SingleOrDefault
- Count
- OrderBy / OrderByDescending
- Min / Max
- Sum / Average

- FromSqlRaw

Plus de détails ici : <http://www.entityframeworktutorial.net/querying-entity-graph-in-entity-framework.aspx>

Tracking

La méthode `SaveChanges()` permet de valider (commit) les modifications dans la base de données, que vous ajoutez, modifiez ou supprimiez des données.

Par défaut, toute requête est réalisée avec suivi (tracking). Entity Framework Core conserve les informations relatives à une instance d'entité dans son traceur de modifications. Ainsi, toutes les modifications détectées dans l'entité sont rendues persistantes dans la base de données en utilisant `SaveChanges()`.

Plus de détails ici : <https://learn.microsoft.com/fr-fr/ef/core/querying/tracking>

Si l'on souhaite empêcher toute modification dans la base de données (données en lecture seule), il suffit de rajouter la ligne `ctx.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;`. Cette ligne désactive ainsi le tracking sur tout le contexte, i.e. toutes les requêtes Linq.

```

    using Microsoft.EntityFrameworkCore;
    using TP2Console.Models.EntityFramework;

    namespace TP2Console
    {
        internal class Program
        {
            static void Main(string[] args)
            {
                using (var ctx = new FilmsDbContext())
                {
                    //Désactivation du tracking => Aucun changement dans la base ne sera effectué.
                    ctx.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

                    //Requête SELECT
                    Film titanic = ctx.Films.First(f => f.Nom.Contains("Titanic"));

                    //Modification de l'entité (dans le contexte seulement)
                    titanic.Description = "Un bateau échoué. Date : " + DateTime.Now;

                    //Sauvegarde du contexte => Application de la modification dans la BD
                    int nbchanges = ctx.SaveChanges();

                    Console.WriteLine("Nombre d'enregistrements modifiés ou ajoutés : " + nbchanges);
                }
            }
        }
    }

```

Tester.

Autre possibilité : utiliser la méthode `AsNoTracking()`, qui donne seulement accès en lecture seule à l'instance (ou les instances) récupérée lors de la requête Linq. Il faudra par contre utiliser cette méthode pour chaque requête.

```

    using Microsoft.EntityFrameworkCore;
    using TP2Console.Models.EntityFramework;

    namespace TP2Console
    {
        internal class Program
        {
            static void Main(string[] args)
            {
                using (var ctx = new FilmsDbContext())
                {
                    //Désactivation du tracking => Aucun changement dans la base ne sera effectué.
                    //ctx.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

                    //Requête SELECT
                    Film titanic = ctx.Films.AsNoTracking().First(f => f.Nom.Contains("Titanic"));

                    //Modification de l'entité (dans le contexte seulement)
                    titanic.Description = "Un bateau échoué. Date : " + DateTime.Now;

                    //Sauvegarde du contexte => Application de la modification dans la BD
                    int nbchanges = ctx.SaveChanges();

                    Console.WriteLine("Nombre d'enregistrements modifiés ou ajoutés : " + nbchanges);
                }
            }
        }
    }

```

Tester.

1.3. Chargement des données

Il existe 3 modes de chargement des données :

- *Chargement explicite* : les données associées sont explicitement chargées à partir de la base de données à un moment ultérieur.
- *Chargement hâtif* : les données associées sont chargées à partir de la base de données dans le cadre de la requête initiale.
- *Chargement différé* (\geq .NET Core 2.1) : les données associées sont chargées de façon transparente à partir de la base de données lors de l'accès à la propriété de navigation.

- Chargement explicite « A la main » :

```

using (var ctx = new FilmsDBContext())
{
    //Chargement de la catégorie Action
    Categorie categorieAction = ctx.Categories.First(c => c.Nom == "Action");
    Console.WriteLine("Categorie : " + categorieAction.Nom);
    Console.WriteLine("Films : ");

    //Chargement des films de la catégorie Action.
    foreach (var film in ctx.Films.Where(f => f.CategorieNavigation.Nom ==
categorieAction.Nom).ToList())
    {
        Console.WriteLine(film.Nom);
    }
}

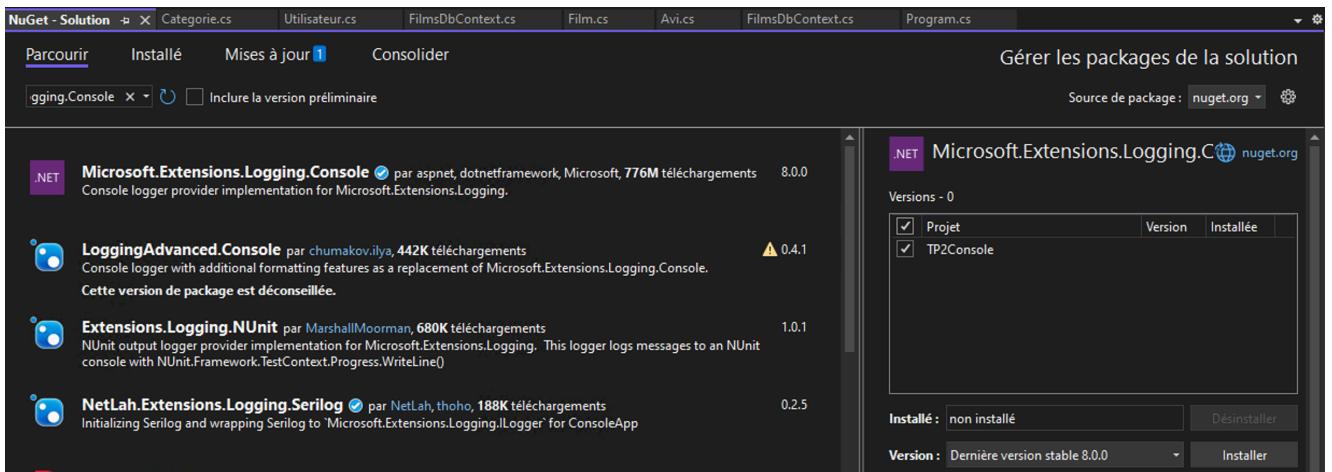
```

Ici, on charge la catégorie "Action", puis on charge « manuellement » les films correspondant à cette catégorie en exécutant une 2^{nde} requête utilisant la propriété de navigation.

Même si son écriture est simple, le chargement explicite à la main n'est pas recommandé. Il est préférable de faire le chargement explicite en utilisant Collection/Reference.

Activer le logger SQL :

- o Installer le package Microsoft.Extensions.Logging.Console :



- Ajouter le code suivant dans la classe de contexte :

```
public static readonly ILoggerFactory MyLoggerFactory = LoggerFactory.Create(builder =>
    builder.AddConsole();

    optionsBuilder.UseLoggerFactory(MyLoggerFactory)
        .EnableSensitiveDataLogging()
        .UseNpgsql("Server=localhost;port=5432;Database=FilmsDB;
uid=postgres; password=postgres;");
```

```
1  using System;
2  using System.Collections.Generic;
3  using Microsoft.EntityFrameworkCore;
4  using Microsoft.Extensions.Logging;
5
6  namespace TP2Console.Models.EntityFramework;
7
8  public partial class FilmsDbContext : DbContext
9  {
10     public FilmsDbContext()
11     {
12     }
13
14     public FilmsDbContext(DbContextOptions<FilmsDbContext> options)
15         : base(options)
16     {
17     }
18
19     public virtual DbSet<Avi> Avis { get; set; }
20
21     public virtual DbSet<Categorie> Categories { get; set; }
22
23     public virtual DbSet<Film> Films { get; set; }
24
25     public virtual DbSet<Utilisateur> Utilisateurs { get; set; }
26
27     public static readonly ILoggerFactory MyLoggerFactory = LoggerFactory.Create(builder => builder.AddConsole());
28
29     protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
30     #warning To protect potentially sensitive information in your connection string, you should move it out of source code. You can avoid so
31     => optionsBuilder.UseLoggerFactory(MyLoggerFactory)
32         .EnableSensitiveDataLogging()
33         .UseNpgsql("Server=localhost;port=5432;Database=FilmsDB; uid=postgres; password=postgres;");
```

- Exécuter l'application. On peut voir les 2 requêtes SQL générées :

```

Console de débogage Microsoft Visual Studio
should only be enabled during development.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
  Executed DbCommand (21ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
  SELECT c.idcategorie, c.description, c.nom
  FROM categorie AS c
  WHERE c.nom = 'Action'
  LIMIT 1
Categorie : Action
Films :
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
  Executed DbCommand (7ms) [Parameters=[@__categorieAction_Nom_0='Action'], CommandType='Text', CommandTimeout='30']
  SELECT f.idfilm, f.description, f.idcategorie, f.nom
  FROM film AS f
  INNER JOIN categorie AS c ON f.idcategorie = c.idcategorie
  WHERE c.nom = @_categorieAction_Nom_0
Volte/Face
Blade Runner
Piège de cristal
58 minutes pour vivre
Pulp fiction
Godzilla
Mission: Impossible
Top Gun
Leon

```

- Chargement explicite (avec Collection et Reference) :

```

using (var ctx = new FilmsDbContext())
{
    Categorie categorieAction = ctx.Catégories.First(c => c.Nom == "Action");
    Console.WriteLine("Categorie : " + categorieAction.Nom);

    //Chargement des films dans categorieAction
    ctx.Entry(categorieAction).Collection(c => c.Films).Load();
    Console.WriteLine("Films : ");
    foreach (var film in categorieAction.Films)
    {
        Console.WriteLine(film.Nom);
    }
}

```

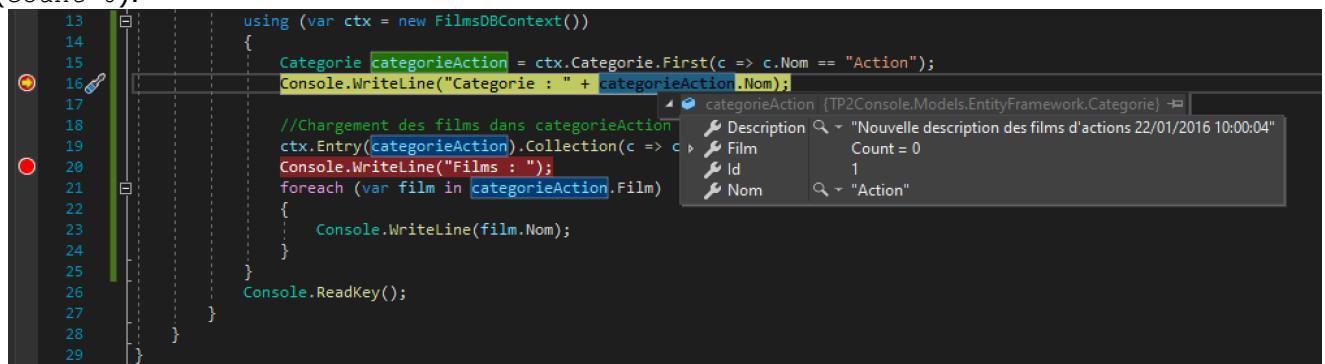
Le chargement explicite d'une propriété de navigation se fait via l'API DbContext.Entry(...). Ici, on charge la catégorie « Action », puis on charge les films dans cette catégorie (via sa propriété de navigation Film) en utilisant la méthode Collection.

Pour mieux comprendre comment cela fonctionne, mettre un point d'arrêt au niveau de chaque ligne Console.WriteLine.

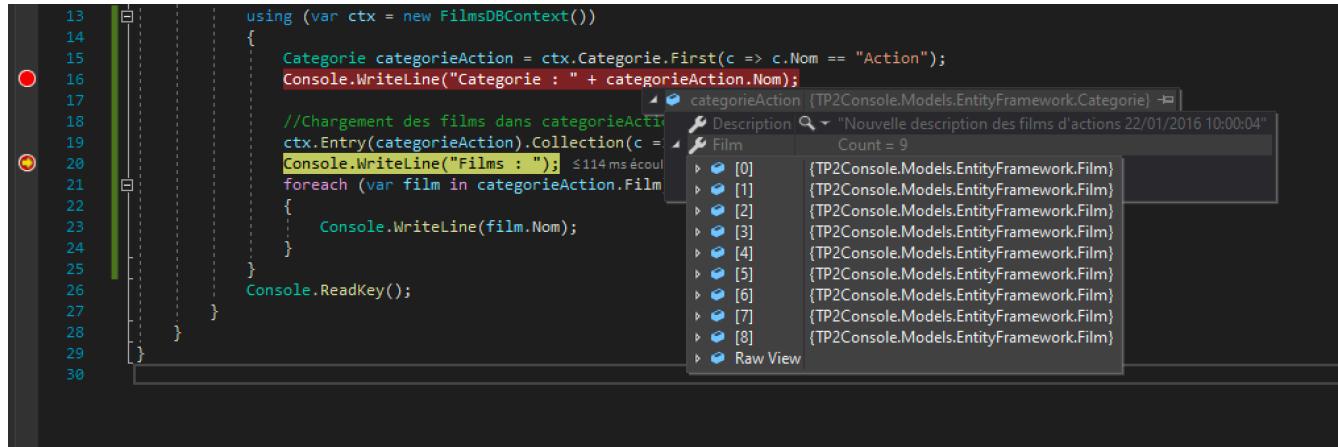
Pour définir un point d'arrêt, il suffit de cliquer dans la barre verticale grise au niveau de la ligne . On peut alors se déplacer dans le code en utilisant :

- Pas à Pas Détailé** : Rentrer dans la fonction
- Pas à Pas Principal** : Aller à la ligne suivante (ne pas rentrer dans la fonction)
- Pas à Pas Sortant** : Aller directement à la fin de la fonction en cours

Au premier point d'arrêt, on remarque que la variable categorieAction ne contient pas de film (count=0).



Après exécution de la ligne ctx.Entry().Collection(), les données des films sont automatiquement chargées dans la variable categorieAction.



Pour faciliter le débogage, on peut aussi utiliser les espions (en plus d'un point d'arrêt) : clic avec le bouton droit de la souris sur la variable > Ajouter un espion.

On voit ensuite les différentes valeurs que prend la variable au fur et à mesure de l'exécution du code.

Espion 1			
Nom	Valeur	Type	
categorieAction	{1: Action}	TP2Console.Models.EntityFramework.Categorie	
Description	"Toto"	string	
Film	Count = 9	System.Collections.Generic.IEnumerable<TP2Console.Models.EntityFramework.Film>	
Id	1	int	
Nom	"Action"	string	

Log :

```
Console de débogage Microsoft Visual Studio
Sensitive data logging is enabled. Log entries and exception messages may include sensitive application data; this mode should only be enabled during development.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (25ms) [Parameters=@p_0='1', CommandType='Text', CommandTimeout='30']
SELECT c.idcategorie, c.description, c.nom
FROM categorie AS c
WHERE c.nom = 'Action'
LIMIT 1
Categorie : Action
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (8ms) [Parameters=@p_0='1', CommandType='Text', CommandTimeout='30']
SELECT f.idfilm, f.description, f.idcategorie, f.nom
FROM film AS f
WHERE f.idcategorie = @p_0
Films :
Volte/Face
Blade Runner
Piege de cristal
58 minutes pour vivre
Pulp fiction
Godzilla
Mission: Impossible
Top Gun
Leon
```

Le chargement explicite est utilisé grâce à 2 méthodes :

- Reference : Quand il s'agit d'une seule entité (Relation one to one).
- Collection : Quand il s'agit d'une collection d'entités (Relation many to many) ; c'est ici le cas.

Exemple avec Reference (à 1 film correspond 1 catégorie) :

```
Film film2 = ctx.Films.First(f => f.Nom == "Titanic");
ctx.Entry(film2).Reference(p => p.CategorieNavigation).Load();
Console.WriteLine("Film : " + film2.Nom + ". Catégorie : " +
film2.CategorieNavigation.Nom);
```

La catégorie est chargée après l'exécution de la 2^{nde} ligne :

```

26     Film film2 = ctx.Film.First(f => f.Nom == "Titanic");
27     ctx.Entry(film2).Load();
28     Console.WriteLine("Avis : " + film2.Navigation.Avis);
29     Console.ReadKey();
30 }
31 }
32 }
33 }
34 }
35 }

```

- Chargement hâtif :

```

using (var ctx = new FilmsDbContext())
{
    //Chargement de la catégorie Action et des films de cette catégorie
    Categorie categorieAction = ctx.Categories
        .Include(c => c.Films)
        .First(c => c.Nom == "Action");
    Console.WriteLine("Categorie : " + categorieAction.Nom);
    Console.WriteLine("Films : ");
    foreach (var film in categorieAction.Films)
    {
        Console.WriteLine(film.Nom);
    }
}

```

Mettre un point d'arrêt sur la ligne `Console.WriteLine`. On peut voir que les films sont bien chargés en même temps que la catégorie.

```

//Chargement hâtif :
using (var ctx = new FilmsDbContext())
{
    //Chargement de la catégorie Action et des films de cette catégorie
    Categorie categorieAction = ctx.Categories
        .Include(c => c.Films)
        .First(c => c.Nom == "Action");
    Console.WriteLine("Categorie : " + categorieAction.Nom);
    Console.WriteLine("Films : ");
    foreach (var film in categorieAction.Films)
    {
        Console.WriteLine(film.Nom);
    }
}

```

Logs :

```

Console de débogage Microsoft Visual Studio
warn: Microsoft.EntityFrameworkCore.Model.Validation[10400]
  Sensitive data logging is enabled. Log entries and exception messages may include sensitive application data; this mode should only be enabled during development.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
  Executed DbCommand (16ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
  SELECT t.idcategorie, t.description, t.nom, f.idfilm, f.description, f.idcategorie, f.nom
  FROM (
    SELECT c.idcategorie, c.description, c.nom
    FROM categorie AS c
    WHERE c.nom = 'Action'
    LIMIT 1
  ) AS t
  LEFT JOIN film AS f ON t.idcategorie = f.idcategorie
  ORDER BY t.idcategorie
Categorie : Action
Films :
Volte/Face
Blade Runner
Piège de cristal
58 minutes pour vivre
Pulp fiction
Godzilla
Mission: Impossible
Top Gun

```

Pour inclure les données avec le chargement hâtif, on utilise la méthode `Include()`. Dans cette méthode il est nécessaire d'indiquer quelles données on veut inclure.

Il est possible d'ajouter plusieurs `Include()` à la fois, si plusieurs propriétés de navigation sont présentes dans la classe.

Sur un modèle de base de données complexe, il arrive parfois qu'une table puisse avoir plusieurs niveaux de données. Dans ce cas, on peut spécifier ce que l'on veut inclure en descendant dans la hiérarchie des liens. Pour effectuer cette opération, on utilisera la méthode `Include()` ainsi que la méthode `ThenInclude()`.

Exemple :

```

using (var ctx = new FilmsDbContext())
{
  //Chargement de la catégorie Action, des films de cette catégorie et des avis
  Categorie categorieAction = ctx.Catégories
    .Include(c => c.Films)
    .ThenInclude(f => f.Avis)
    .First(c => c.Nom == "Action");
}

```

Il est possible de faire appel à plusieurs `ThenInclude()` pour continuer à inclure les niveaux de données associées suivants.

Logs :

```

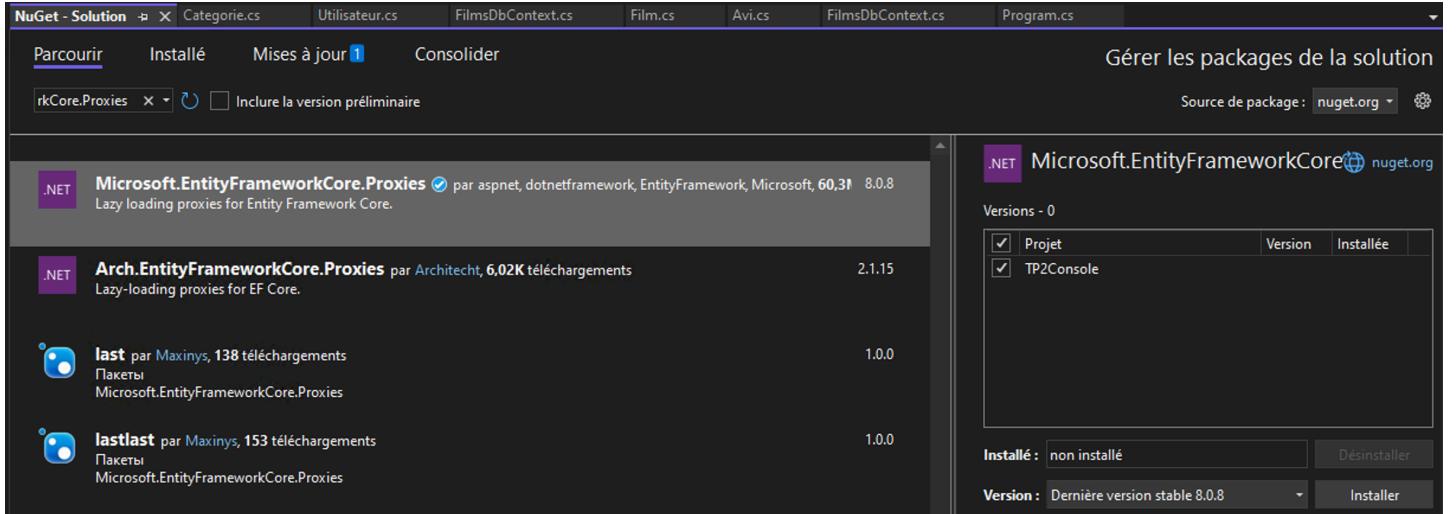
Console de débogage Microsoft Visual Studio
warn: Microsoft.EntityFrameworkCore.Model.Validation[10400]
  Sensitive data logging is enabled. Log entries and exception messages may include sensitive application data; this mode should only be enabled during development.
warn: Microsoft.EntityFrameworkCore.Query[20504]
  Compiling a query which loads related collections for more than one collection navigation, either via 'Include' or through projection, but no 'QuerySplittingBehavior' has been configured. By default, Entity Framework will use 'QuerySplittingBehavior.SingleQuery', which can potentially result in slow query performance. See https://go.microsoft.com/fwlink/?linkid=2134277 for more information. To identify the query that's triggering this warning call 'ConfigureWarnings(w => w.Throw(RelationalEventId.MultipleCollectionIncludeWarning))'.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
  Executed DbCommand (22ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
  SELECT t.idcategorie, t.description, t.nom, t0.idfilm, t0.description, t0.nom, t0.idfilm0, t0.idutilisateur, t0.commentaire, t0.note
  FROM (
    SELECT c.idcategorie, c.description, c.nom
    FROM categorie AS c
    WHERE c.nom = 'Action'
    LIMIT 1
  ) AS t
  LEFT JOIN (
    SELECT f.idfilm, f.description, f.idcategorie, f.nom, a.idfilm AS idfilm0, a.idutilisateur, a.commentaire, a.note
    FROM film AS f
    LEFT JOIN avis AS a ON f.idfilm = a.idfilm
  ) AS t0 ON t.idcategorie = t0.idcategorie
  ORDER BY t.idcategorie, t0.idfilm, t0.idfilm0

```

- Chargement différé :

Le chargement différé (ou paresseux) des données (Lazy loading) est un mode dans lequel la récupération de données de la base de données est différée jusqu'à ce qu'elle soit réellement nécessaire. Cela semble être une bonne chose et, dans certains scénarios, cela peut aider à améliorer les performances d'une application. Dans d'autres scénarios, cela peut dégrader considérablement les performances, en particulier dans les applications Web. Pour cette raison, le chargement différé a été introduit dans EF Core 2.1 en tant que fonctionnalité devant être installée manuellement.

Pour le chargement différé, il est nécessaire de disposer au minimum de .NET Core 2.1 et d'installer la **dernière version** du package NuGet `Microsoft.EntityFrameworkCore.Proxies` :



Il faut ensuite modifier la méthode `OnConfiguring()` du contexte et ajouter un appel à la méthode `UseLazyLoadingProxies()`.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    // To protect potentially sensitive information in your connection string, you should move it out of source code. You can avoid scaffolding the connection string by
    // using the EntityDataSourceConfigurationFilterAttribute or the EntityDataSourceFilterSetting on your context configuration
    // optionsBuilder.UseLoggerFactory(MyLoggerFactory)
    //     .EnableSensitiveDataLogging()
    //     .UseNpgsql("Server=localhost;port=5432;Database=FilmsDB; uid=postgres; password=postgres;")
    //     .UseLazyLoadingProxies();
}
```

Pour définir les attributs qui seront chargés en différé, il est nécessaire d'ajouter le mot clé `virtual` devant les attributs de navigation dans les classes du modèle. En mode Reverse Engineering, cela a déjà été fait :

- o Classe `Film` : `public virtual Categorie CategorieNavigation { get; set; }`
- o Classe `Categorie` : `public virtual ICollection<Film> Films { get; set; }`

Ici, nous allons l'appliquer entre `Categorie` et `Film`, il est donc nécessaire de modifier la classe `Categorie` :

- o Ajouter le constructeur paramétré suivant :


```
private ILazyLoader _lazyLoader;
public Categorie(ILazyLoader lazyLoader)
{
    _lazyLoader = lazyLoader;
}
```
- oModifier la property permettant d'accéder aux films ainsi (mettre en commentaires le code actuel de la property) :


```
private ICollection<Film> films; //Doit être écrit de la même façon que la property Films mais en minuscule
```

```
[InverseProperty("CategorieNavigation")]
public virtual ICollection<Film> Films
{
    get
    {
        return _lazyLoader.Load(this, ref films);
    }
    set { films = value; }
}
```

Code méthode `Main` :

```
using (var ctx = new FilmsDbContext())
{
    //Chargement de la catégorie Action
    Categorie categorieAction = ctx.Categories.First(c => c.Nom == "Action");
```

```

Console.WriteLine("Categorie : " + categorieAction.Nom);
Console.WriteLine("Films : ");

    //Chargement des films de la catégorie Action.
    foreach (var film in categorieAction.Films) // lazy loading initiated
    {
        Console.WriteLine(film.Nom);
    }
}

```

Logs :

```

Sensitive data logging is enabled. Log entries and exception messages may include sensitive application data; this mode should only be enabled during development.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (21ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT c.idcategorie, c.description, c.nom
      FROM categorie AS c
      WHERE c.nom = 'Action'
      LIMIT 1
Categorie : Action
Films :
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (7ms) [Parameters=[@__p_0='1'], CommandType='Text', CommandTimeout='30']
      SELECT f.idfilm, f.description, f.idcategorie, f.nom
      FROM film AS f
      WHERE f.idcategorie = @_p_0
Volte/Face
Blade Runner
Piege de cristal
58 minutes pour vivre
Pulp fiction
Godzilla
Mission: Impossible
Top Gun
Leon

```

La trace correspond à celle du chargement explicite « à la main », sauf qu'ici à aucun moment le chargement des films n'est codé explicitement (pas de `ctx.Films.Where(f => f.CategorieNavigation.Nom == categorieAction.Nom)`).

Plus de détails ici :

- o <https://learn.microsoft.com/fr-fr/ef/core/querying/related-data>
- o <https://www.learnentityframeworkcore.com/lazy-loading>

Désactiver le chargement différé (mettre en commentaire les modifications précédentes).

Bilan

Il est important de comprendre ces différents mécanismes et de savoir quand les utiliser, sous peine d'avoir de très mauvaises performances lorsque l'on travaille sur de gros volumes de données.

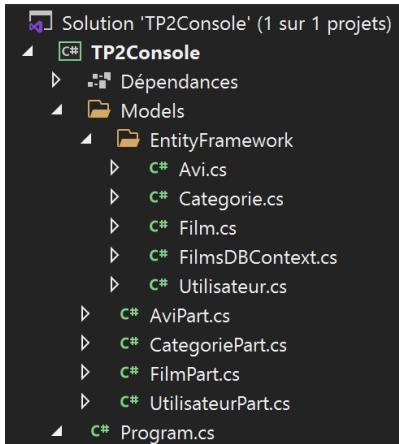
Il est en général conseillé de ne pas utiliser le chargement paresseux sauf si vous êtes certain que c'est la meilleure solution. C'est pourquoi (contrairement aux versions précédentes d'EF), le chargement différé n'est pas activé par défaut dans Entity Framework Core. Selon les cas, utilisez soit le chargement hâtif, soit le chargement explicite (avec Collection et Reference).

Quand on est sûr de travailler sur des éléments liés, il est important de les charger en avance.

2. Partie Pratique

2.1. Exercice 1 : Extensions des classes

1. Dans le dossier `Models`, rajouter 4 classes partielles (correspondant aux 4 classes du modèle). Convention de nommage : `<XXX>Part.cs` (Par exemple, `AviPart.cs`). Attention à mettre le même namespace que celui contenant les classes générées afin qu'elles puissent compléter les autres classes partielles générées par EF.



2. Rajouter la méthode `ToString()` à ces classes (en codant, par exemple, un retour "id : nom")

2.2. Exercice 2 : Sélection des données

Le but de cette partie est d'afficher dans la console les résultats. Pour toutes les questions ci-dessous, il faudra créer une fonction (on nommera les fonctions `Exo2q<XXX>` qui seront appelées dans le `main`). Quand il est indiqué d'afficher un objet sans préciser quelle propriété, on utilisera la méthode `ToString()` précédemment créée.

Note : on pourra utiliser au choix la syntaxe lambda ou SQL.

1. Exemple : Afficher tous les films.

```
static void Main(string[] args)
{
    Exo2Q1();
    Console.ReadKey();

}

public static void Exo2Q1()
{
    var ctx = new FilmsDBContext();
    foreach (var film in ctx.Films)
    {
        Console.WriteLine(film.ToString());
    }
}

//Autre possibilité :
public static void Exo2Q1Bis()
{
    var ctx = new FilmsDBContext();

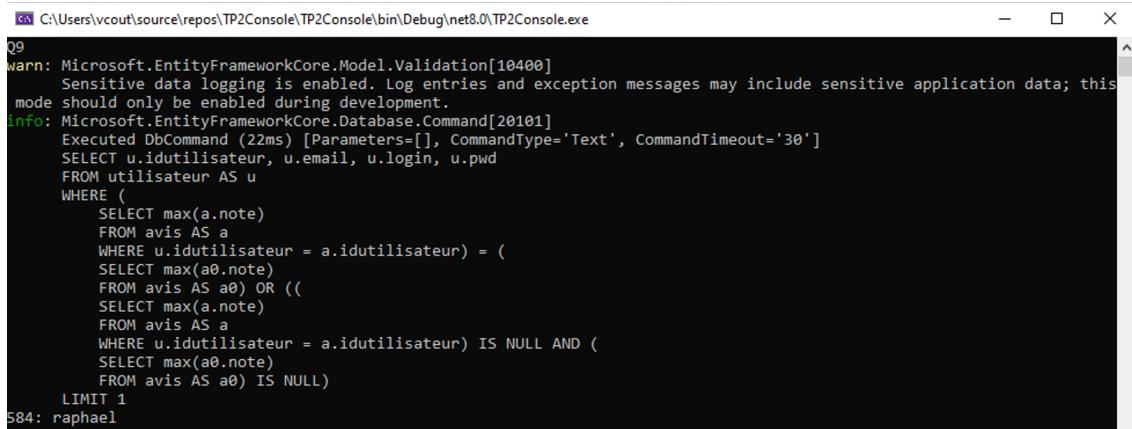
    //Pour que cela marche, il faut que la requête envoie les mêmes noms de colonnes que les
    classes c#.
    var films = ctx.Films.FromSqlRaw("SELECT * FROM film");

    foreach (var film in films)
    {
        Console.WriteLine(film.ToString());
    }
}
```

Pour les questions suivantes, ne pas utiliser `FromSqlRaw`. Normalement, vous n'aurez pas non plus besoin de `Include` (pas de chargement hâtif), si vous procédez toujours de la façon suivante :

Il faut toujours partir du `DbSet` duquel on souhaite récupérer les données. Ex. « Afficher tous les films qui... » -> `ctx.Filmsxxxxxx`

2. Afficher les emails de tous les utilisateurs.
 3. Afficher tous les utilisateurs triés par login croissant.
 4. Afficher les noms et id des films de la catégorie « Action ».
 5. Afficher le nombre de catégories.
 6. Afficher la note la plus basse dans la base.
 7. Rechercher tous les films qui commencent par « le » (pas de respect de la casse => 14 résultats).
- Remarque : on peut le faire de 2 façons différentes au moins :*
- <https://stackoverflow.com/questions/45708715/entity-framework-ef-functions-like-vs-string-contains>
8. Afficher la note moyenne du film « Pulp Fiction » (note : le nom du film ne devra pas être sensible à la casse).
 9. Afficher l'utilisateur qui a mis la meilleure note dans la base (on pourra le faire en 2 instructions, mais essayer de le faire en une seule).



```
C:\Users\vcout\source\repos\TP2Console\TP2Console\bin\Debug\net8.0\TP2Console.exe
99
warn: Microsoft.EntityFrameworkCore.Model.Validation[10400]
      Sensitive data logging is enabled. Log entries and exception messages may include sensitive application data; this mode should only be enabled during development.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (22ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT u.idutilisateur, u.email, u.login, u.pwd
      FROM utilisateur AS u
      WHERE (
          SELECT max(a.note)
          FROM avis AS a
          WHERE u.idutilisateur = a.idutilisateur) = (
          SELECT max(a0.note)
          FROM avis AS a0) OR (((
          SELECT max(a.note)
          FROM avis AS a
          WHERE u.idutilisateur = a.idutilisateur) IS NULL AND (
          SELECT max(a0.note)
          FROM avis AS a0) IS NULL)
      LIMIT 1
584: raphael
```

2.3. Exercice 3 : Modification des données

Chaque question est indépendante, à coder dans des méthodes différentes.

Documentation : <http://www.entityframeworktutorial.net/efcore/saving-data-in-connected-scenario-in-ef-core.aspx>

Ajoutez-vous en tant qu'utilisateur

L'ajout se fait en 3 étapes :

1. Création et initialisation de l'objet
2. Ajout au contexte. Pour ajouter au contexte, il suffit de l'ajouter à la collection Utilisateur
3. Sauvegarde du contexte

Modifier un film

Rajouter une description au film « L'armee des douze singes » et le mettre dans la catégorie « Drame ».

Supprimer un film

Supprimer le film « L'armee des douze singes ».

Note : il n'y a pas de delete cascade sur la foreign key. Penser à supprimer les Avis associés !

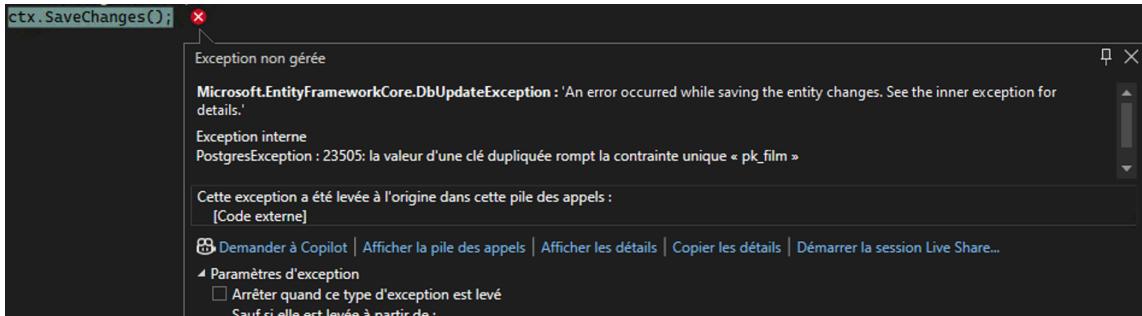
Ajouter un avis

Ajouter vos avis et note à votre film préféré (ou détesté).

Ajouter 2 films dans la catégorie « Drame ».

Utiliser `AddRange()`.

Attention, à l'exécution vous aurez l'erreur suivante :



Dans les scripts, nous n'utilisons pas les séquences lors des insertions, mais fixons l'ID. Nous essayons d'insérer ici un ID déjà existant. Il faut donc exécuter les lignes suivantes, permettant de modifier la valeur actuelle de la séquence de la table des films puis relancer l'ajout :

```
ALTER TABLE film ENABLE TRIGGER ALL;
SELECT pg_catalog.setval(pg_get_serial_sequence('film', 'idfilm'), MAX(idfilm)) FROM film;
```

3. Informations supplémentaires

- Méthode SaveChanges()

Il est possible de regrouper plusieurs opérations (modification, ajout et/ou suppression) et les committer en une seule fois (un seul appel à SaveChanges()). Dans ce cas, si une modification échoue, toutes les modifications échoueront.

```
using (var ctx = new FilmsDbContext())
{
    Categorie categorieAction = ctx.Catégories.First(c => c.Nom == "Action");
    categorieAction.Description = "Toto";

    ctx.Utilisateurs.Add(new Utilisateur
    {
        Login = "Login1",
        Pwd = "Pwd1",
        Email = "login1@gmoil.com"
    });

    int nbchanges = ctx.SaveChanges();

    Console.WriteLine("Nombre d'enregistrements modifiés ou ajoutés : " + nbchanges);
}
```

- Gestion de la concurrence

Entity Framework Core gère automatiquement les cas de conflits lors de la modification concomitante d'enregistrements. À chaque exécution de la méthode SaveChanges, EF vérifie si la valeur d'origine du jeton d'accès concurrentiel est la même que la valeur lue en base de données.

Il y a deux méthodes pour définir des jetons de concurrence :

- Annotation : [ConcurrencyCheck]
- API Fluent : on utilise la méthode IsConcurrencyToken()

Dans le cas où un conflit est détecté, SaveChanges renvoie une exception de type DbUpdateConcurrencyException : <https://learn.microsoft.com/fr-fr/ef/core/saving/concurrency>, <https://learn.microsoft.com/fr-fr/ef/core/modeling/concurrency>

On peut aussi utiliser un horodatage à la place du jeton d'accès concurrentiel. L'horodatage consiste à modifier un champ qui contient la date courante (timestamp) à chaque insertion ou modification de ligne.

Il y a deux méthodes pour définir un timestamp :

- Annotation : [Timestamp]
- API Fluent : On utilise la méthode IsRowVersion().

<https://learn.microsoft.com/fr-fr/ef/core/modeling/concurrency>

- Méthodes asynchrones

Nous avons utilisé les méthodes synchrones dans les exercices précédents. EF Core prend en charge la programmation asynchrone et propose des méthodes asynchrones pour le requêtage, la modification et l'ajout d'enregistrements :

- SingleAsync()
- FindAsync()
- AddAsync()
- AddRangeAsync()
- SaveChangesAsync()