

Presque toutes les applications accèdent à des bases de données. Écrire manuellement les classes d'accès aux données (`DataAccess`, `Service`) est une tâche longue, source d'erreurs, répétitive et donc peu intéressante pour le développeur.

Aujourd'hui, de nombreux outils permettent de simplifier l'écriture des couches d'accès aux données : les ORM (Object Relational Mapping ou Mapping objet-relationnel).

Les ORM proposent en général 2 modes de fonctionnement :

- Le fonctionnement « Code First » : le développeur écrit le code (la partie "objet") avec des annotations particulières (spécifique à l'ORM utilisé) ou le définit via une API spécifique, et génère ensuite la base de données à partir du code (Object  $\Rightarrow$  Relational).
- Le fonctionnement « Database First », qui permet de générer les objets à partir de la base de données.

Nous allons, dans ce TP, utiliser Entity Framework Core en mode « Database First ». Depuis la version Core, ce mode est appelé « Reverse Engineering » et n'est pas le mode préconisé par Microsoft. Il convient cependant bien quand il s'agit de créer un nouveau projet avec une base de données déjà existante.

*Remarque : nous verrons le mode « Code first » dans le TP suivant.*

## **1. Mise en place et bases du fonctionnement**

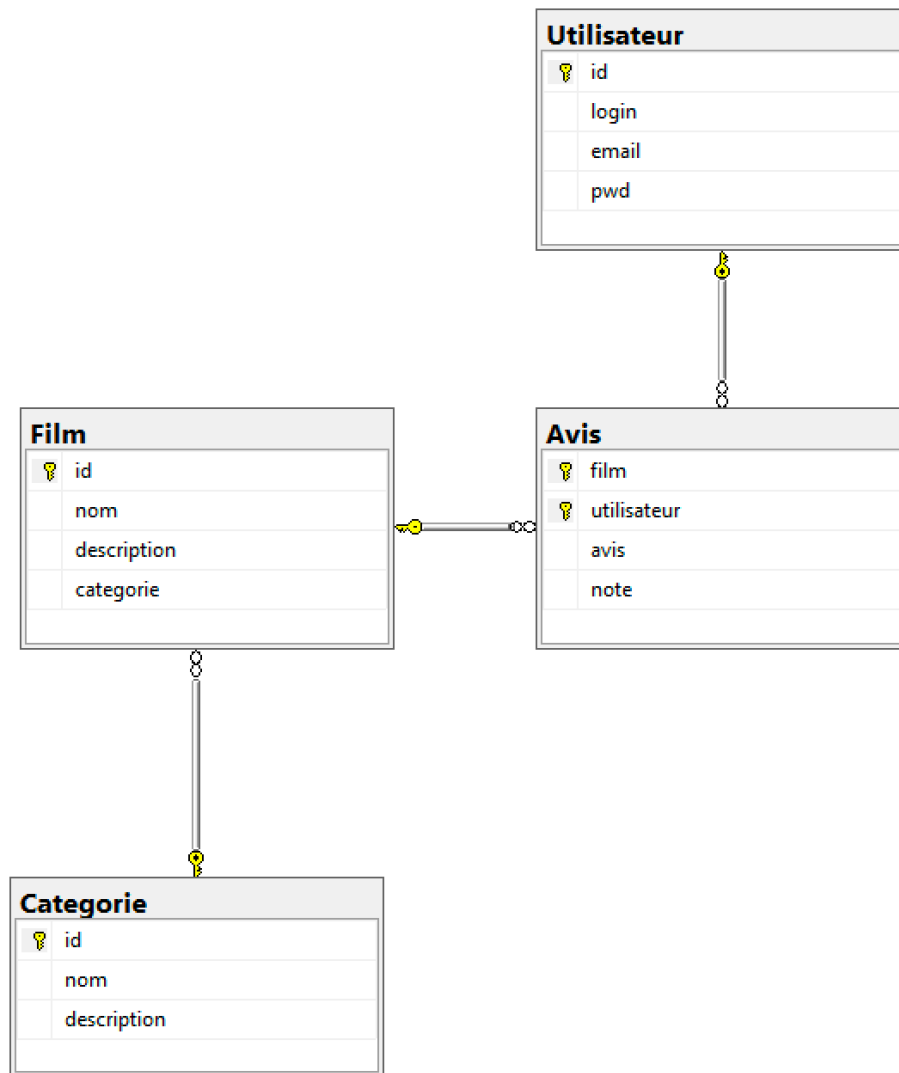
### 1.1. Mise en place

a) Création de la base de données sous PostgreSQL.

Sous PgAdmin 4, créer une base `FilmsDB` puis exécuter le script `scriptFilmsPostgreSQL.sql`.

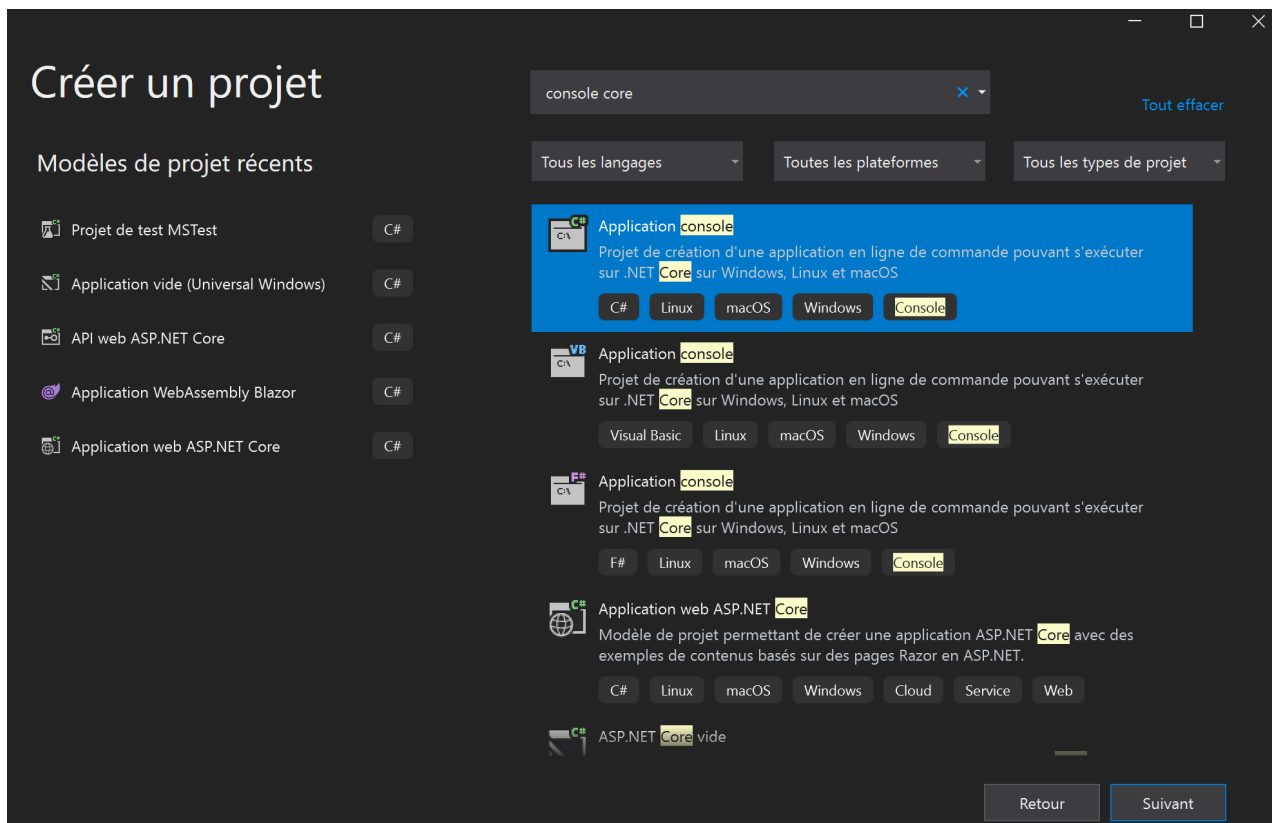
*Remarque : Dans la VM, mot de passe root PostgreSQL : `postgres`*

Schéma de la base de données :



#### b) Création du projet .NET

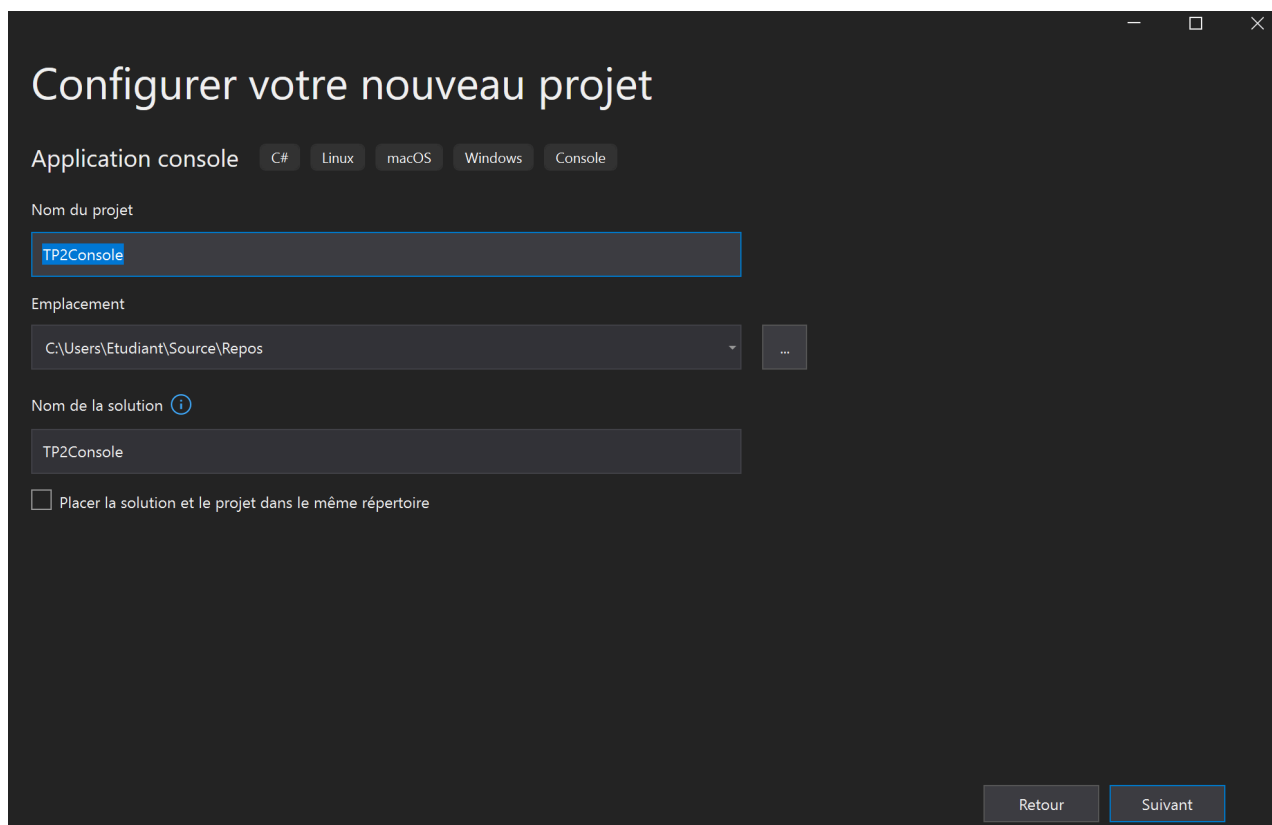
Pour simplifier, nous allons travailler sur une application console. Lancer Visual Studio et créer une nouvelle *Application console* (**.NET Core**) nommée **TP2console**.



## Modèles de projet récents

The screenshot shows the 'New Project' dialog in Visual Studio. At the top, there are filters for 'console core' (with a close button), 'Tous les langages' (All languages), 'Toutes les plateformes' (All platforms), and 'Tous les types de projet' (All project types). The 'ASP.NET Core Web Application' template is selected and highlighted in blue. Below the template name, it says 'Projet de création d'une application en ligne de commande pouvant s'exécuter sur .NET Core sur Windows, Linux et macOS'. The language is set to 'C#', and the platform is 'Console'. Other visible templates include 'Application console' (C#), 'Application console' (VB), and 'Application console' (F#). The 'ASP.NET Core Web Application' template description mentions 'Modèle de projet permettant de créer une application ASP.NET Core avec des exemples de contenus basés sur des pages Razor en ASP.NET.' The language 'C#' is selected, and the platform 'Web' is chosen.

Retour Suivant



Application console C# Linux macOS Windows Console

Nom du projet

TP2Console

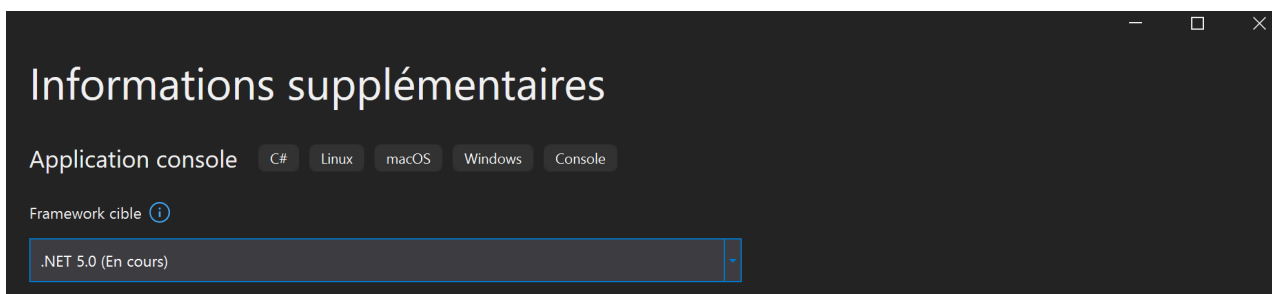
Emplacement

Nom de la solution ⓘ

TP2Console

☐ Placer la solution et le projet dans le même répertoire

Retour Suivant



Application console C# Linux macOS Windows Console

Framework cible ⓘ

.NET 5.0 (En cours)

### 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 Core (par exemple, la version 5.0.x sur .NET Core 5).**

*Le scaffolding (échafaudage, en français) est une technique dont l'objectif est la génération de code. Plus exactement, il s'agit de générer une structure de base sur laquelle de nouveaux éléments peuvent se greffer, en s'appuyant sur une spécification. Pour le cas d'Entity Framework Core avec une approche Database First, le scaffolding permet la génération d'un modèle objet à partir d'une source de données existante et de nombreux paramètres dont l'objectif est la personnalisation du code généré.*

*Packages NuGet à installer pour :*

- **SQL Server :** `Microsoft.EntityFrameworkCore.SqlServer`, `Microsoft.EntityFrameworkCore.SqlServer.Design`
- **MySQL :** `Pomelo.EntityFrameworkCore.MySQL`, `Pomelo.EntityFrameworkCore.MySQL.Design`
- **+ dans tous les cas :** `Microsoft.EntityFrameworkCore.Tools`

Pour migrer le modèle, nous allons utiliser la *Console du gestionnaire de package* de Visual Studio.

Pour cela, il faut utiliser le menu *Outils > Gestionnaire de package nuget > Console du gestionnaire de package*.

Une fois la console ouverte, on peut écrire des commandes PowerShell. Pour migrer la base de données, nous allons utiliser la commande `Scaffold-DbContext`.

```
Scaffold-DbContext [-Connection] <String>
                  [-Provider] <String>
                  [-OutputDir <String>]
                  [-Context <String>]
                  [-Schemas <String>]
                  [-Tables <String>]
                  [-DataAnnotations]
                  [-Force]
                  [-Project <String>]
                  [-StartupProject <String>]
                  [-Environment <String>]
                  [<CommonParameters>]
```

Principaux arguments :

`-Connection`

Représente la chaîne de connexion complète qui doit être utilisée pour lire les informations de la source de données. Le nom de l'argument peut être omis lorsque sa valeur est la première dans la chaîne d'arguments.

`-Provider`

La valeur associée à cet argument indique le nom complet du fournisseur de données qui doit être utilisé pour accéder à la source de données.

`-OutputDir`

Cet argument permet de spécifier le répertoire dans lequel les fichiers générés doivent être placés. Si ce répertoire n'existe pas, il est créé lors de l'opération.

`-Context`

Le nom d'une base de données n'est pas forcément en adéquation avec les conventions de nommage utilisées par le code .NET. Il n'est pas rare de trouver une base de données dont le nom est entièrement en majuscules, par exemple. Les contextes de données générés ayant par défaut le nom de la source de données accolé au suffixe `Context`, cela peut être gênant. Cette problématique est réglée par l'utilisation de l'argument `-Context` qui permet de définir le nom du type de contexte généré.

-Schemas

Lorsque les tables de la base de données n'appartiennent pas toutes au schéma de l'utilisateur connecté, cet argument permet de spécifier les noms des schémas pour lesquels doivent être générés des types d'entités. Les noms de schémas fournis doivent être séparés par des virgules :  
schema1,schema2,...

-Tables

Permet de filtrer la liste des tables pour lesquelles du code doit être généré. Comme pour les schémas, les noms des tables concernées sont séparés par des virgules.

-DataAnnotations

L'argument -DataAnnotations indique que le code généré devrait utiliser les data annotations tant que possible. Lorsqu'il n'est pas présent, la configuration des types d'entités générée n'utilise que l'API Fluent.

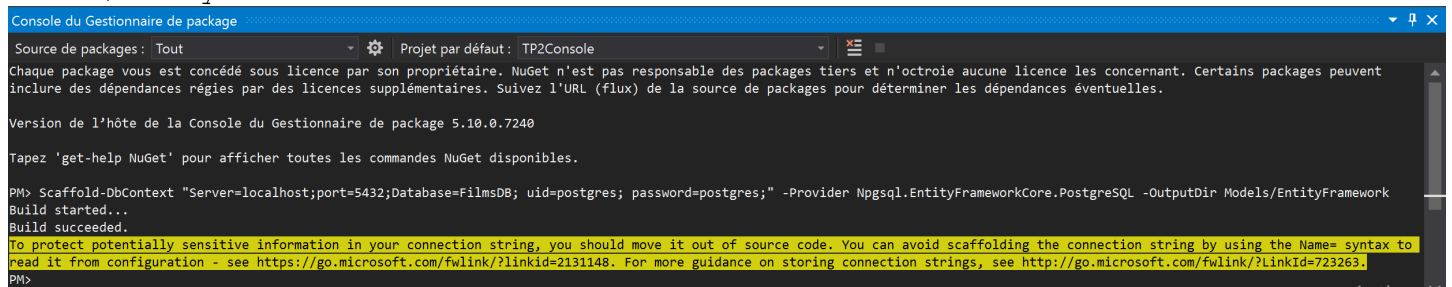
Dans Entity Framework Core, il y a donc 2 méthodes pour créer les modèles :

- Les data annotations, si l'on ajoute le paramètre -DataAnnotations.
- L'API Fluent.

### 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
```



The screenshot shows the Package Manager Console in Visual Studio. The command executed is: `Scaffold-DbContext "Server=localhost;port=5432;Database=FilmsDB; uid=postgres; password=postgres;" -Provider Npgsql.EntityFrameworkCore.PostgreSQL -OutputDir Models/EntityFramework`. The output shows the build starting and succeeding. A warning message is also visible: "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."

**Sous Visual Studio pour Mac, il faut d'abord installer dotnet ef, puis la commande devient (à exécuter dans un terminal) :**

```
dotnet tool install --global dotnet-ef
```

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

Plus de détails ici : <https://docs.microsoft.com/fr-fr/ef/core/miscellaneous/cli/dotnet>

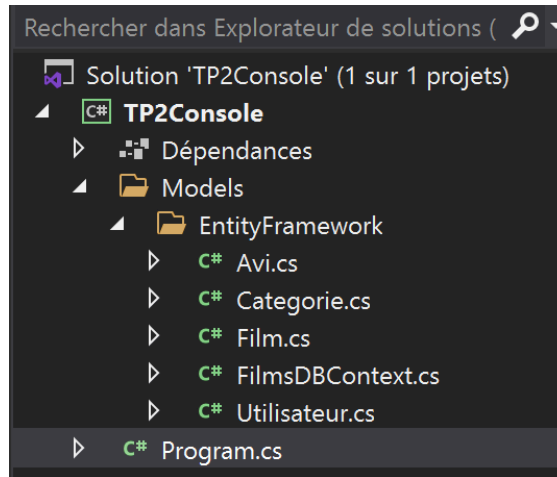
*Dans le cas de MySQL, la commande devient, par exemple :*

```
Scaffold-DbContext "Server=localhost;port=3306;Database=filmsdb;uid=root;password=xxxx;" -Provider Pomelo.EntityFrameworkCore.MySQL -OutputDir Models/EntityFramework
```

*Dans SQL Server :*

```
Scaffold-DbContext "Server=localhost;Database=FilmsDB; Trusted_Connection=True;" -Provider Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models/EntityFramework
```

Une fois cette commande lancée, les classes correspondant à la base de données seront générées automatiquement dans Models/EntityFramework.



Chacun de ces fichiers de code correspond à une table de la source de données, à l'exception du fichier `FilmsDBContext.cs`, qui contient le contexte de données.

Contenu du fichier `Avi.cs` :

```
namespace TP2Console.Models.EntityFramework
{
    6 références
    public partial class Avi
    {
        3 références
        public int Film { get; set; }
        3 références
        public int Utilisateur { get; set; }
        1 référence
        public string Avis { get; set; }
        1 référence
        public decimal Note { get; set; }

        1 référence
        public virtual Film FilmNavigation { get; set; }
        1 référence
        public virtual Utilisateur UtilisateurNavigation { get; set; }
    }
}
```

On remarque 2 propriétés de navigation, créées à partir des clés étrangères, permettant d'accéder aux objets associés.

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` :

```

namespace TP2Console.Models.EntityFramework
{
    6 références
    public partial class Film
    {
        0 références
        public Film()
        {
            Avis = new HashSet<Avi>();
        }

        1 référence
        public int Id { get; set; }
        1 référence
        public string Nom { get; set; }
        1 référence
        public string Description { get; set; }
        2 références
        public int Categorie { get; set; }

        1 référence
        public virtual Categorie CategorieNavigation { get; set; }
        2 références
        public virtual ICollection<Avi> Avis { get; set; }
    }
}

```

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.

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://docs.microsoft.com/en-us/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 propriétés. 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)
{
    if (!optionsBuilder.IsConfigured)
    {
        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.ToTable("film");
        entity.Property(e => e.Id).HasColumnName("id");
        entity.Property(e => e.Categorie).HasColumnName("categorie");
        entity.Property(e => e.Description).HasColumnName("description");
        entity.Property(e => e.Nom)
            .IsRequired()
    }
    )

```

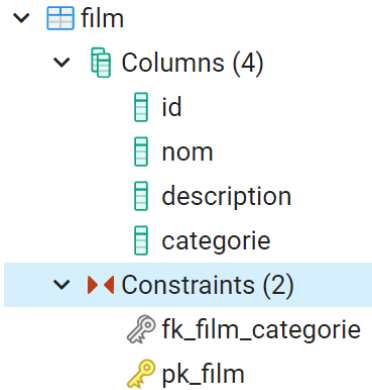
```

        .HasMaxLength(50)
        .HasColumnName("nom");

entity.HasOne(d => d.CategorieNavigation)
    .WithMany(p => p.Films)
    .HasForeignKey(d => d.Categorie)
    .OnDelete(DeleteBehavior.ClientSetNull)
    .HasConstraintName("fk_film_categorie");
});

```

Rappel : définition de la table :



La property `Id` (`e.Id`) de la classe correspond à la colonne `id` de la table. Il n'y a pas d'autres caractéristiques définies ici car les types correspondent : `int` (de la property)  $\leftrightarrow$  `int` ou `integer` (type de colonne de la table).

Pour la property `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 property `Nom`, on remarque que `NOT NULL` dans la base de données correspond à `IsRequired` et que le nombre de caractères maximum de la chaîne est précisé (`HasMaxLength`).

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

```

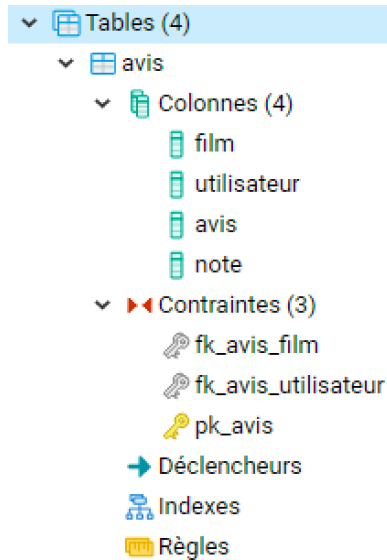
modelBuilder.Entity<Avi>(entity =>
{
    entity.HasKey(e => new { e.Film, e.Utilisateur })
        .HasName("pk_avis");

    entity.ToTable("avis");
    entity.Property(e => e.Film).HasColumnName("film");
    entity.Property(e => e.Utilisateur).HasColumnName("utilisateur");
    entity.Property(e => e.Avis).HasColumnName("avis");
    entity.Property(e => e.Note).HasColumnName("note");
    entity.HasOne(d => d.FilmNavigation)
        .WithMany(p => p.Avis)
        .HasForeignKey(d => d.Film)
        .OnDelete(DeleteBehavior.ClientSetNull)
        .HasConstraintName("fk_avis_film");
    entity.HasOne(d => d.UtilisateurNavigation)
        .WithMany(p => p.Avis)
        .HasForeignKey(d => d.Utilisateur)
        .OnDelete(DeleteBehavior.ClientSetNull)
        .HasConstraintName("fk_avis_utilisateur");
});

```

Rappel : définition de la table :





`HasKey` permet de définir la clé primaire et `HasName` son nom. On remarque qu'il n'y a pas de clé primaire définie dans les autres entités. En réalité, elles sont définies implicitement car les champs se nomment `id` dans les tables (autre possibilité : les nommer `TableNameId`). Si vous utilisez un nom de colonne primary key différent dans les tables, EF Core générera une méthode `HasKey`.

```

entity.HasOne(d => d.FilmNavigation)
    .WithMany(p => p.Avis)
    .HasForeignKey(d => d.Film)
    .OnDelete(DeleteBehavior.ClientSetNull)
    .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` (`FilmNavigation`) (`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` (`Film`), le type de suppression des enregistrements liés `.OnDelete(DeleteBehavior.ClientSetNull)` (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
<b>Cascade</b>	Suppression	Suppression
<b>ClientSetNull</b> (Par défaut)	Valeur à NULL	Aucun
<b>SetNull</b>	Valeur à NULL	Valeur à NULL
<b>Restrict</b>	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
<b>Cascade</b> (Par défaut)	Suppression	Suppression
<b>ClientSetNull</b>	Lève une exception	Aucun
<b>SetNull</b>	Lève une exception	Lève une exception
<b>Restrict</b>	Aucun	Aucun

Le code que nous venons d'analyser correspond à l'API Fluent : <http://www.entityframeworktutorial.net/efcore/fluent-api-in-entity-framework-core.aspx>, <https://www.learnentityframeworkcore.com/configuration/fluent-api>

*En mode Code First, ce sera à vous de créer ce code.*

## Création du modèle en utilisant les Data Annotations

Supprimer le dossier `Models` et 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
```

### Sous macOS (Terminal) :

```
dotnet ef DbContext Scaffold "Server=localhost;port=5432;Database=FilmsDB; uid=postgres; password=postgres;" Npgsql.EntityFrameworkCore.PostgreSQL -o "Models/EntityFramework" -d
```

Classe `Avi` générée :

```
namespace TP2Console.Models.EntityFramework
{
    [Table("avis")]
    8 références
    public partial class Avi
    {
        [Key]
        [Column("film")]
        3 références
        public int Film { get; set; }
        [Key]
        [Column("utilisateur")]
        3 références
        public int Utilisateur { get; set; }
        [Column("avis")]
        0 références
        public string Avis { get; set; }
        [Column("note")]
        0 références
        public decimal Note { get; set; }

        [ForeignKey(nameof(Film))]
        [InverseProperty("Avis")]
        2 références
        public virtual Film FilmNavigation { get; set; }
        [ForeignKey(nameof(Utilisateur))]
        [InverseProperty("Avis")]
        2 références
        public virtual Utilisateur UtilisateurNavigation { get; set; }
    }
}
```

L'annotation `[InverseProperty]` permet de définir la propriété « inverse » d'une clé étrangère (Cf. explication plus loin).

Classe `Film` :

```

namespace TP2Console.Models.EntityFramework
{
    [Table("film")]
    7 références
    public partial class Film
    {
        0 références
        public Film()
        {
            Avis = new HashSet<Avi>();
        }

        [Key]
        [Column("id")]
        0 références
        public int Id { get; set; }
        [Required]
        [Column("nom")]
        [StringLength(50)]
        0 références
        public string Nom { get; set; }
        [Column("description")]
        0 références
        public string Description { get; set; }
        [Column("categorie")]
        2 références
        public int Categorie { get; set; }

        [ForeignKey(nameof(Categorie))]
        [InverseProperty("Films")]
        2 références
        public virtual Categorie CategorieNavigation { get; set; }
        [InverseProperty(nameof(Avi.FilmNavigation))]
        2 références
        public virtual ICollection<Avi> Avis { get; set; }
    }
}

```

#### 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>
- [ForeignKey(nameof(MyProperty))] : Permet de spécifier le nom de la property qui est clé étrangère dans la relation. ForeignKey est toujours associée à l'instruction InverseProperty qui permet d'indiquer la property liée dans la classe associée.

Exemple dans Avi :

```

[ForeignKey(nameof(Film))]
[InverseProperty("Avis")]
public virtual Film FilmNavigation { get; set; }

```

A comparer dans Film à :

```

[InverseProperty(nameof(Avi.FilmNavigation))]
public virtual ICollection<Avi> Avis { get; set; }

```

nameof n'est pas obligatoire. On peut aussi écrire : [InverseProperty("FilmNavigation")]

Plus d'explications ici :

<http://www.entityframeworktutorial.net/code-first/inverseproperty-dataannotations-attribute-in-code-first.aspx>  
<https://stackoverflow.com/questions/40480601/what-is-difference-between-inverse-property-and-foreign-key-in-entity-framework>

- [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*. 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://www.entityframeworktutorial.net/efcore/fluent-api-in-entity-framework-core.aspx>

Classe de contexte :

```

0 références
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasAnnotation("Relational:Collation", "French_France.1252");

    modelBuilder.Entity<Avis>(entity =>
    {
        entity.HasKey(e => new { e.Film, e.Utilisateur })
            .HasName("pk_avis");

        entity.HasOne(d => d.FilmNavigation)
            .WithMany(p => p.Avis)
            .HasForeignKey(d => d.Film)
            .OnDelete(DeleteBehavior.ClientSetNull)
            .HasConstraintName("fk_avis_film");

        entity.HasOne(d => d.UtilisateurNavigation)
            .WithMany(p => p.Avis)
            .HasForeignKey(d => d.Utilisateur)
            .OnDelete(DeleteBehavior.ClientSetNull)
            .HasConstraintName("fk_avis_utilisateur");
    });

    modelBuilder.Entity<Film>(entity =>
    {
        entity.HasOne(d => d.CategorieNavigation)
            .WithMany(p => p.Films)
            .HasForeignKey(d => d.Categorie)
            .OnDelete(DeleteBehavior.ClientSetNull)
            .HasConstraintName("fk_film_categorie");
    });

    OnModelCreatingPartial(modelBuilder);
}

```

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

`HasAnnotation` permet d'ajouter des attributs d'annotation à l'entité, selon un couple (*annotation, valeur*). Ici, la collation est fixée à la valeur « French\_France.1252 » (WIN1252 : encodage spécifique à Windows).

*Rappel : la collation permet de définir le comportement du SGBD lors du traitement des chaînes de caractères notamment lors des opérations de comparaison et de tri. Collation PostgreSQL :*

<https://www.postgresql.org/docs/current/multibyte.html>,

<https://www.postgresql.org/docs/13/collation.html>

Le reste du code n'est dédié qu'à la création de la clé primaire composée de 2 champs et des FK.

#### Explications de la clé étrangère suivante :

```

modelBuilder.Entity<Avis>(entity =>
{
    ...
    entity.HasOne(d => d.FilmNavigation)
        .WithMany(p => p.Avis)
        .HasForeignKey(d => d.Film)
        .OnDelete(DeleteBehavior.ClientSetNull)
        .HasConstraintName("fk_avis_film");
    ...
}

```

- `entity.HasOne(d => d.FilmNavigation)` : Propriété de navigation de la classe `Avis` (table `Avis` -> `Film`). `HasOne` car un seul film lié à l'avis. On remarque `[InverseProperty("Avis")]` devant la propriété de navigation `FilmNavigation` de la classe `Avis` permettant de définir la propriété de navigation liée dans la table `Film`.
- `.WithMany(p => p.Avis)` : Propriété de navigation de la classe `Film` (`WithMany` car collection de `<Avis>`). On remarque `[InverseProperty("FilmNavigation")]` devant la propriété de navigation

Avis de la classe `Avi` permettant de faire la liaison entre les 2 propriétés de navigation. `InverseProperty` permet de définir une "référence croisée".

- `.HasForeignKey(d => d.Film)` : Property de `Avi` qui est FK.
- 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://docs.microsoft.com/fr-fr/ef/core/modeling/relationships?tabs=fluent-api%2Cfluent-api-simple-key%2Csimple-key>

#### d) Modification du modèle

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.

```

Categorie.cs  Film.cs  Avi.cs  FilmsDBContext.cs  Program.cs  Utilisateur.cs
C# TP2Console  TP2Console.Program  Main(string[] ar
1  using System;
2  using TP2Console.Models.EntityFramework;
3  using System.Linq;
4
5  namespace TP2Console
6  {
7      0 références
8      class Program
9      {
10         0 références
11         static void Main(string[] args)
12         {
13             //Requête SELECT
14             Film titanic = ctx.Films.First(f => f.Nom.Contains("Titanic"));
15
16             //Modification de l'entité (dans le contexte seulement)
17             titanic.Description = "Un bateau échoué. Date : " + DateTime.Now;
18
19             //Sauvegarde du contexte => Application de la modification dans la BD
20             int nbchanges = ctx.SaveChanges();
21
22             Console.WriteLine("Nombre d'enregistrements modifiés ou ajoutés : " + nbchanges);
23         }
24         Console.ReadKey();
25     }
26 }
27

```

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

Résultat dans la base de données :

Data Output	Explain	Messages	Notifications
id [PK] integer	nom character varying (50)	description text	categorie integer
1	3 Titanic	Un bateau échoué. Date : 20/09/2021 22:22:40	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 ( $s \Rightarrow s...$ ). 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 :  
<http://www.technicaloverload.com/linq-single-vs-singleordefault-vs-first-vs-firstordefault/>
- 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 ajoutiez, 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://docs.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.

```

1  using System;
2  using TP2Console.Models.EntityFramework;
3  using System.Linq;
4  using Microsoft.EntityFrameworkCore;
5
6  namespace TP2Console
7  {
8      0 références
9      class Program
10     {
11         0 références
12         static void Main(string[] args)
13         {
14             using (var ctx = new FilmsDBContext())
15             {
16                 //Désactivation du tracking => Aucun changement dans la base ne sera effectué
17                 ctx.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
18
19                 //Requête SELECT
20                 Film titanic = ctx.Films.First(f => f.Nom.Contains("Titanic"));
21
22                 //Modification de l'entité (dans le contexte seulement)
23                 titanic.Description = "Un bateau échoué. Date : " + DateTime.Now;
24
25                 //Sauvegarde du contexte => Application de la modification dans la BD
26                 int nbchanges = ctx.SaveChanges();
27
28                 Console.WriteLine("Nombre d'enregistrements modifiés ou ajoutés : " + nbchanges);
29             }
30             Console.ReadKey();
31         }
32     }
33 }

```

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.

```

1  using System;
2  using TP2Console.Models.EntityFramework;
3  using System.Linq;
4  using Microsoft.EntityFrameworkCore;
5
6  namespace TP2Console
7  {
8      0 références
9      class Program
10     {
11         0 références
12         static void Main(string[] args)
13         {
14             using (var ctx = new FilmsDBContext())
15             {
16                 //Désactivation du tracking => Aucun changement dans la base ne sera effectué
17                 //ctx.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
18
19                 //Requête SELECT
20                 Film titanic = ctx.Films.AsNoTracking().First(f => f.Nom.Contains("Titanic"));
21
22                 //Modification de l'entité (dans le contexte seulement)
23                 titanic.Description = "Un bateau échoué. Date : " + DateTime.Now;
24
25                 //Sauvegarde du contexte => Application de la modification dans la BD
26                 int nbchanges = ctx.SaveChanges();
27
28                 Console.WriteLine("Nombre d'enregistrements modifiés ou ajoutés : " + nbchanges);
29             }
30             Console.ReadKey();
31         }
32     }
33 }

```

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

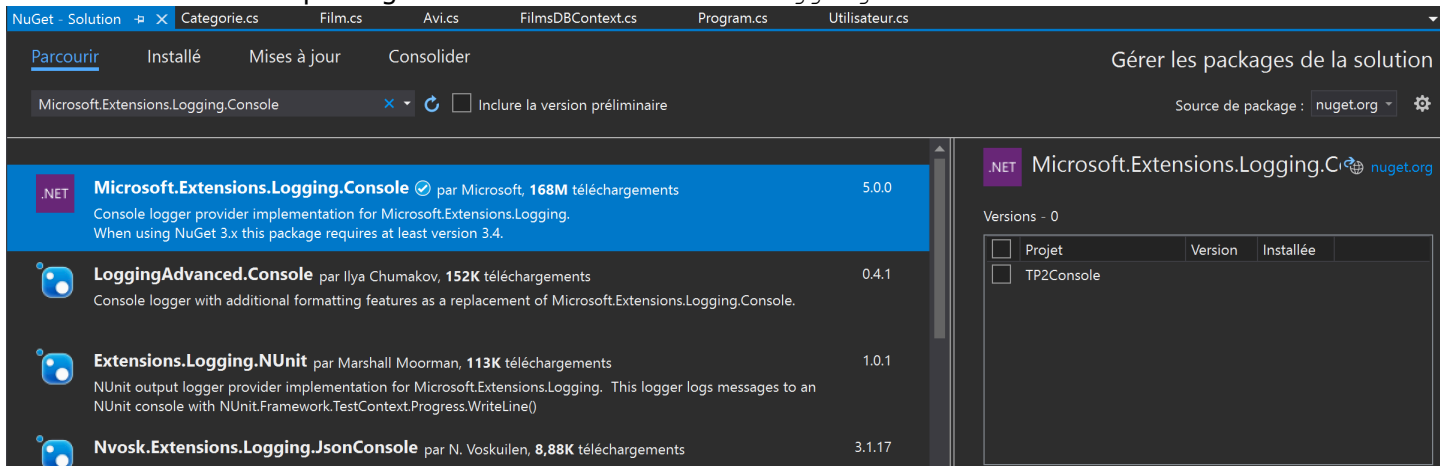
    //Chargement des films de la catégorie Action.
    foreach (var film in ctx.Films.Where(f => f.CategorieNavigation.Nom ==
categorieAction.Nom).ToList())
    {
        Console.WriteLine(film.Nom);
    }
}
```

Ici, on charge la catégorie "Action", puis on charge « manuellement » les films correspondant à cette catégorie en exécutant une 2<sup>de</sup> 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 :

- 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;");
```

```

NuGet - Solution  Categorie.cs  Film.cs  Avi.cs  FilmsDBContext.cs*  Program.cs  Utilisateur.cs
TP2Console
4  using Microsoft.Extensions.Logging;
5
6  #nullable disable
7
8  namespace TP2Console.Models.EntityFramework
9  {
10     4 références
11     public partial class FilmsDBContext : DbContext
12     {
13         public static readonly ILoggerFactory MyLoggerFactory = LoggerFactory.Create(builder => builder.AddConsole());
14
15         1 référence
16         public FilmsDBContext() {}
17
18         0 références
19         public FilmsDBContext(DbContextOptions<FilmsDBContext> options)
20             : base(options){}
21
22         0 références
23         public virtual DbSet<Avi> Avis { get; set; }
24         1 référence
25         public virtual DbSet<Categorie> Categories { get; set; }
26         1 référence
27         public virtual DbSet<Film> Films { get; set; }
28         0 références
29         public virtual DbSet<Utilisateur> Utilisateurs { get; set; }
30
31         0 références
32         protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
33         {
34             if (!optionsBuilder.IsConfigured)
35             {
36                 #warning: To protect potentially sensitive information in your connection string, you should move it out of source code. You can a
37                 optionsBuilder.UseLoggerFactory(MyLoggerFactory)
38                     .EnableSensitiveDataLogging()
39                     .UseNpgsql("Server=localhost;port=5432;Database=FilmsDB; uid=postgres; password=postgres;");
40             }
41         }
42     }

```

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

```

C:\Users\Etudiant\Source\Repos\TP2Console\TP2Console\bin\Debug\net5.0\TP2Console.exe
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.Infrastructure[10403]
      Entity Framework Core 5.0.10 initialized 'FilmsDBContext' using provider 'Npgsql.EntityFrameworkCore.PostgreSQL' w
      ith options: SensitiveDataLoggingEnabled
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (5ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT c.id, 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 (10ms) [Parameters=[@__categorieAction_Nom_0='Action'], CommandType='Text', CommandTimeout='30']
      SELECT f.id, f.categorie, f.description, f.nom
      FROM film AS f
      INNER JOIN categorie AS c ON f.categorie = c.id
      WHERE c.nom = @__categorieAction_Nom_0
Volte/Face
Blade Runner
Piege 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.Categories.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);
    }
}

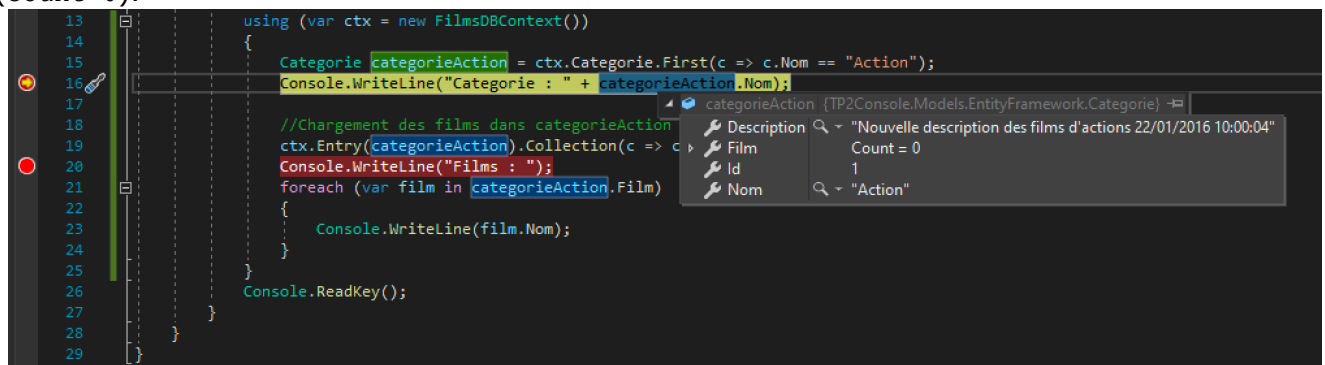
```

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`.

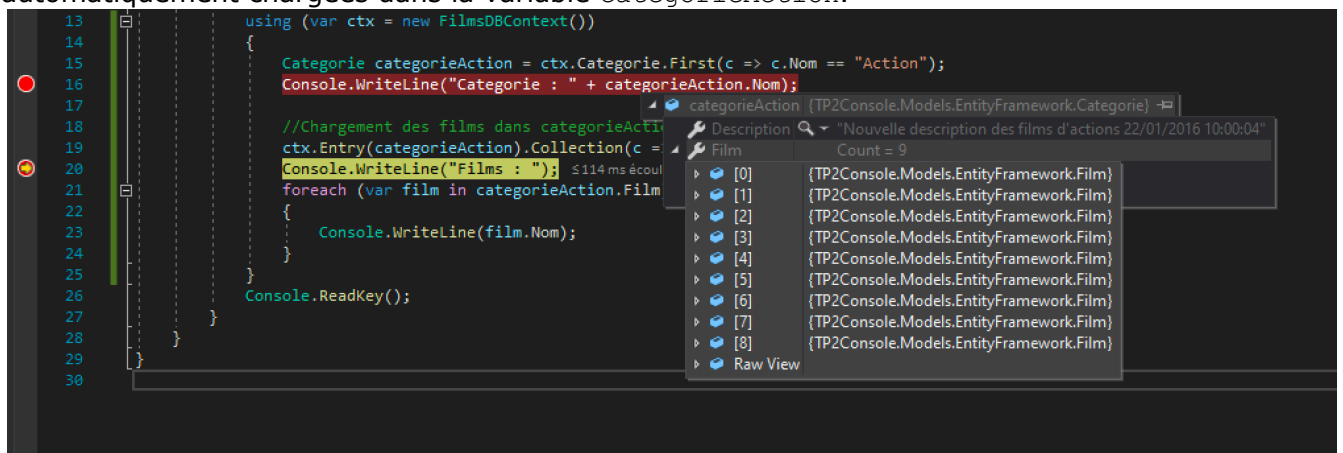
Pour définir un point d'arrêt, il suffit de cliquer dans la barre verticale grise au niveau de la ligne. On peut alors se déplacer dans le code en utilisant :

- Pas à Pas Détaillé  : 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
 <code>categorieAction</code>	{1: Action}	TP2Console.Models.Ent
 Description	"Toto"	string
 Film	Count = 9	System.Collections.Gene
 Id	1	int
 Nom	"Action"	string

Log :

```

C:\Users\Etudiant\Source\Repos\TP2Console\TP2Console\bin\Debug\net5.0\TP2Console.exe
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.Infrastructure[10403]
      Entity Framework Core 5.0.10 initialized 'FilmsDBContext' using provider 'Npgsql.EntityFrameworkCore.PostgreSQL' w
      ith options: SensitiveDataLoggingEnabled
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (4ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT c.id, c.description, c.nom
      FROM categorie AS c
      WHERE c.nom = 'Action'
      LIMIT 1
Categorie : Action
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (25ms) [Parameters=[@__p_0='1'], CommandType='Text', CommandTimeout='30']
      SELECT f.id, f.categorie, f.description, f.nom
      FROM film AS f
      WHERE f.categorie = @__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<sup>de</sup> ligne :

```

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

```

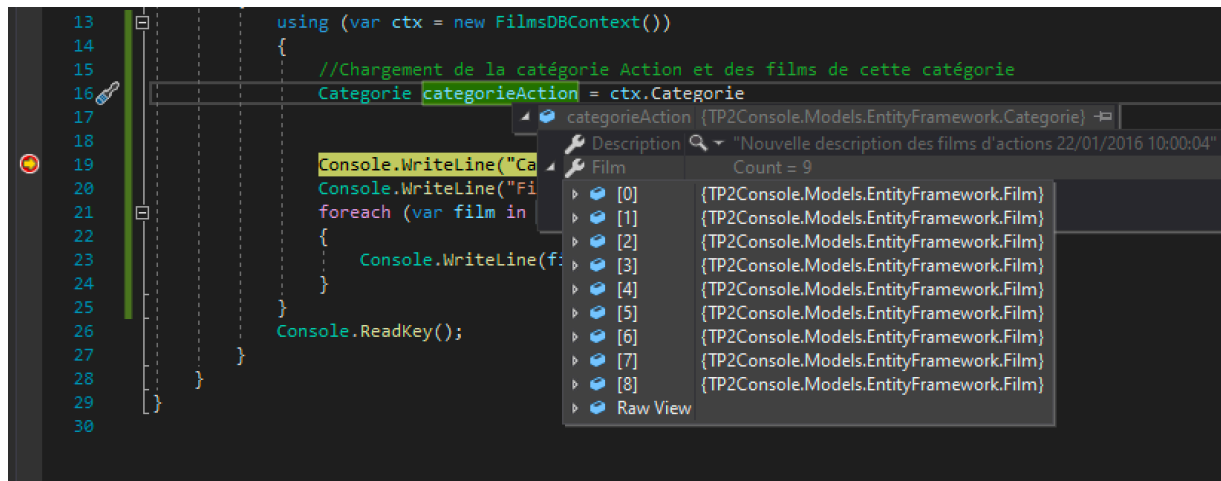
- Chargement hâtif :

```

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.



Logs :

```

C:\Users\Etudiant\Source\Repos\TP2Console\TP2Console\bin\Debug\net5.0\TP2Console.exe
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.Infrastructure[10403]
      Entity Framework Core 5.0.10 initialized 'FilmsDbContext' using provider 'Npgsql.EntityFrameworkCore.PostgreSQL' w
      ith options: SensitiveDataLoggingEnabled
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (6ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT t.id, t.description, t.nom, f.id, f.categorie, f.description, f.nom
      FROM (
        SELECT c.id, c.description, c.nom
        FROM categorie AS c
        WHERE c.nom = 'Action'
        LIMIT 1
      ) AS t
      LEFT JOIN film AS f ON t.id = f.categorie
      ORDER BY t.id, f.id
Categorie : Action
Films :
Volte/Face
Blade Runner
Piege de cristal
58 minutes pour vivre
Pulp fiction
Godzilla
Mission: Impossible
Top Gun
Leon
  
```

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.Categories
        .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 :

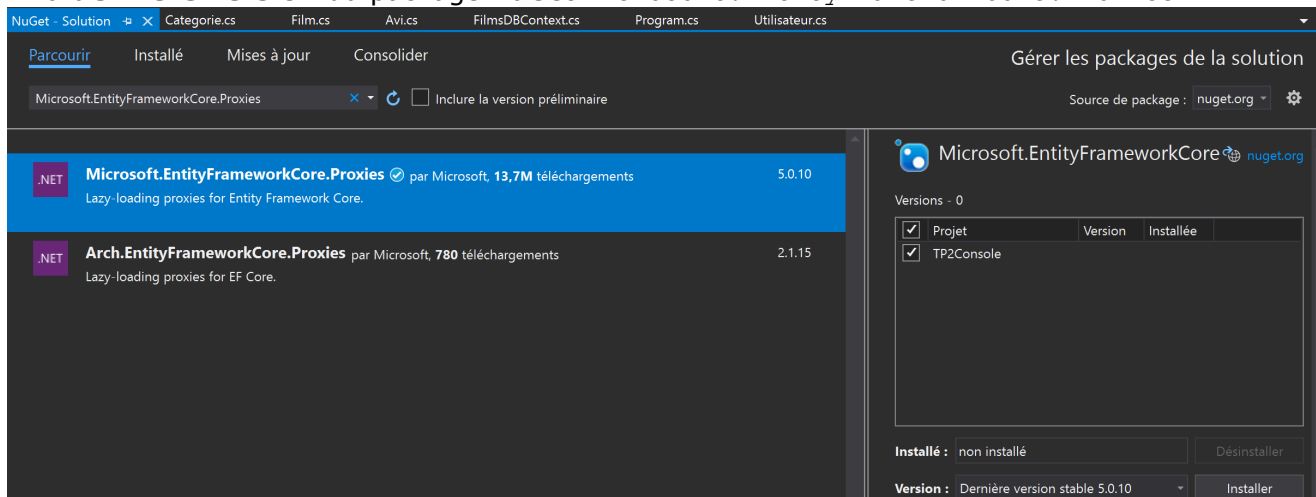
```

C:\Users\Etudiant\Source\Repos\TP2Console\TP2Console\bin\Debug\net5.0\TP2Console.exe
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.Infrastructure[10403]
      Entity Framework Core 5.0.10 initialized 'FilmsDbContext' using provider 'Npgsql.EntityFrameworkCore.PostgreSQL' w
      ith options: SensitiveDataLoggingEnabled
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 'QuerySplit
      tingBehavior.SingleQuery' which can potentially result in slow query performance. See https://go.microsoft.com/fwlink/?l
      inkid=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 (9ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT t.id, t.description, t.nom, t0.id, t0.categorie, t0.description, t0.nom, t0.film, t0.utilisateur, t0.avis,
      t0.note
      FROM (
        SELECT c.id, c.description, c.nom
        FROM categorie AS c
        WHERE c.nom = 'Action'
        LIMIT 1
      ) AS t
      LEFT JOIN (
        SELECT f.id, f.categorie, f.description, f.nom, a.film, a.utilisateur, a.avis, a.note
        FROM film AS f
        LEFT JOIN avis AS a ON f.id = a.film
      ) AS t0 ON t.id = t0.categorie
      ORDER BY t.id, t0.id, t0.film, t0.utilisateur

```

- **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)
{
    if (!optionsBuilder.IsConfigured)
    {
        // To protect potentially sensitive information in your connection string, you should move it out of source code. See http://go.microsoft.com/fwlink/?linkid=2134277 for more information.
        optionsBuilder.UseLoggerFactory(MyLoggerFactory)
            .EnableSensitiveDataLogging()
            .UseNpgsql("Server=localhost;port=5432;Database=FilmsDB; uid=postgres; password=postgres;");
        optionsBuilder.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;
}
```

- o Modifier 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 :

```
C:\Users\Etudiant\Source\Repos\TP2Console\TP2Console\bin\Debug\net5.0\TP2Console.exe
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.Infrastructure[10403]
      Entity Framework Core 5.0.10 initialized 'FilmsDBContext' using provider 'Npgsql.EntityFrameworkCore.PostgreSQL' w
      ith options: SensitiveDataLoggingEnabled using lazy-loading proxies
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (4ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT c.id, 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 (5ms) [Parameters=[@__p_0='1'], CommandType='Text', CommandTimeout='30']
      SELECT f.id, f.categorie, f.description, f.nom
      FROM film AS f
      WHERE f.categorie = @__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 :

- <https://docs.microsoft.com/fr-fr/ef/core/querying/related-data>
- <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.

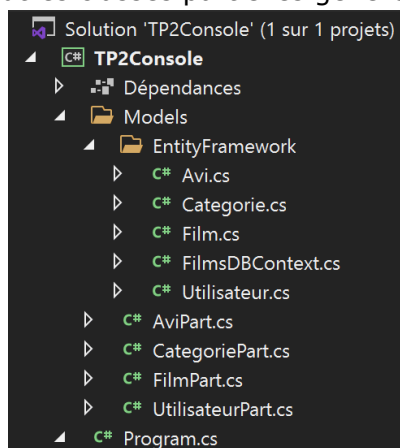
Il est en général conseillé de ne pas utiliser le chargement paresseux sauf si vous êtes certain que c'est la meilleure solution. C'est pourquoi (contrairement aux versions précédentes d'EF), le chargement différé n'est pas activé par défaut dans Entity Framework Core. Selon les cas, utilisez soit le 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`). Voir explication section d) partie 1. 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).**

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).
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\Etudiant\Source\Repos\TP2Console\TP2Console\bin\Debug\net5.0\TP2Console.exe
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.Infrastructure[10403]
      Entity Framework Core 5.0.10 initialized 'FilmsDBContext' using provider 'Npgsql.EntityFrameworkCore.PostgreSQL' w
ith options: SensitiveDataLoggingEnabled using lazy-loading proxies
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (29ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT u.id, u.email, u.login, u.pwd
      FROM utilisateur AS u
      WHERE ((
        SELECT MAX(a.note)
        FROM avis AS a
        WHERE u.id = a.utilisateur) = (
        SELECT MAX(a0.note)
        FROM avis AS a0)) OR (((
        SELECT MAX(a.note)
        FROM avis AS a
        WHERE u.id = a.utilisateur) 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 !*

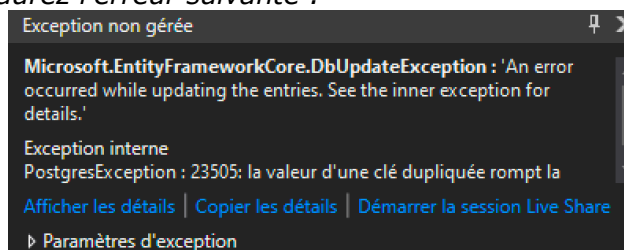
### Ajouter un avis

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 :



Cette erreur est connue sous PostgreSQL et est due au type `serial` et au fait qu'on envoie plusieurs lignes en même temps. Pour la corriger, exécuter la commande SQL suivante :

```
ALTER TABLE film ENABLE TRIGGER ALL;
```

```
SELECT pg_catalog.setval(pg_get_serial_sequence('film', 'id'), MAX(id)) FROM film;
```

## 3. Informations supplémentaires

- Méthode `SaveChanges()`

Il est possible de regrouper plusieurs opérations (modification, ajout et/ou suppression) et les commiter 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.Categories.First(c => c.Nom == "Action");
    categorieAction.Description = "Toto";

    ctx.Utilisateurs.Add(new Utilisateur
    {
        Login = "Login1",
        Pwd = "Pwd1",
        Email = "login1@gmail.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. A 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://docs.microsoft.com/fr-fr/ef/core/saving/concurrency>, <https://docs.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://docs.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()`