

---

# Entity Framework Core



## 1. Qu'est ce qu'un ORM ?

### 1. Entity Framework ?

- Historique
- Les différentes approches
- Comment ça marche ?

### 1. Mise en place

#### 3.1. Les packages Nuget

#### 3.2. Conventions, Fluent API et Data-

Annotations

#### 3.3. Fluent-API Best Practices

#### 3.4. Configuration des classes

#### 3.5. La classe DbContext

#### 3.6. La Connection String

#### 3.7. Les DbSet<T>

#### 3.8. Héritage

## 1. Les Commandes

- Migration

## 5. Seed Data

## 6. Utilisation de DbContext

### 6.1. Create

### 6.2. Read

### 6.3. Update

### 6.4. Delete

### 6.5. Exception

## 7. Les relations

### 7.1. One-to-One

### 7.2. Many-to-One

### 7.3. Many-to-Many

### 7.4 OnDelete

### 7.5 Données relationnelles

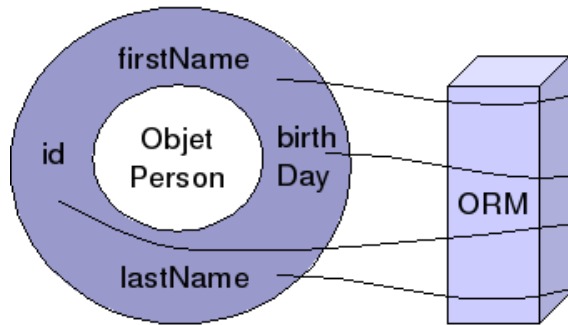
## 8. Références

# Table des matières

---

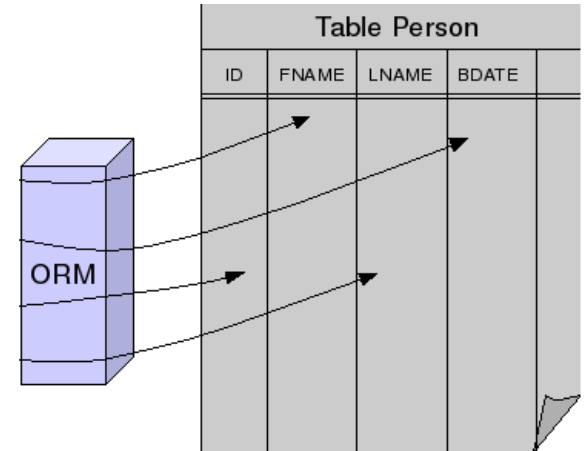
# 1. Qu'est ce qu'un ORM ?

# ORM pour Objet-Relationnel-Model.



Il s'agit d'une couche d'abstraction entre les données et l'application (C# dans notre cas).

Le mapping objet-relationnel consiste à déclarer une association entre une (ou plusieurs) classe et une table, et chaque attribut de la classe avec un champ de la table associée. Par exemple, la classe Person sera associée à la table Person



# Les avantages

- L'utilisation d'un ORM nous permet de nous abstraire du SGBD. Pour passer d'un sgbd à un autre, nous avons juste une connection string à modifier.
- Il empêche la redondance de code. (Le **CRUD** de chaque entité se ressemble étrangement)  
Nous gagnons donc en productivité.
- Très utile sur des projets de petites envergures.
- Nous permet d'éviter d'écrire de longues requête SQL au sein de notre code.

# Les inconvénients

- Pas de totale maîtrise de ce qui se passe
- Devient vite gourmand en ressource si les requêtes sont trop complexes.
- N'est pas forcément capable d'utiliser des concepts avancés des SGBD

## 2. Entity Framework ?

Entity Framework



**EF Core** est la solution ORM préconisée par Microsoft et parfaitement intégrée à l'environnement .Net.

# Historique

Année	Version	Framework	Améliorations notables
2008	Entity Framework	.Net 3.5 SP1	
2010	Entity Framework 4.0	.Net 4.0	
2011	Entity Framework 4.1	.Net 4.0	Apparition de l'approche code-first
2012	Entity Framework 4.3.1		Support de la migration
2012	Entity Framework 5	.Net 4.5	
2013	EF 6		Open-source et disponible sur GitHub
2017	EF Core 2.0	.Net Core 2.0	Multi-plateforme(Linux, Windows, MacOS)
2019	EF Core 3.0	.Net Core 3.0	Couplage fort avec C# 8.0

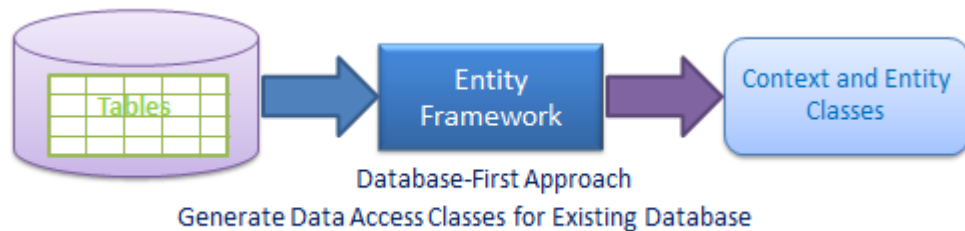


## 2. Entity Framework ?

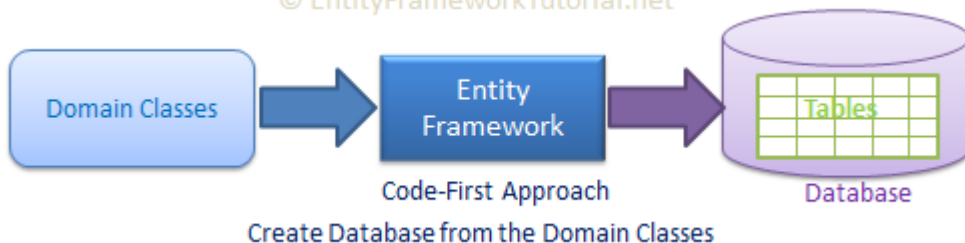
- Parfaitement intégré à DotNet.
- Utilisation simple de requêtes LinQ
- Facile à mettre en place
- Permet l'utilisation d'un grand nombre de SGBD (Sql Server, Oracle, MySql, .... )

## 2. Entity Framework ?

### Différentes approches



© EntityFrameworkTutorial.net

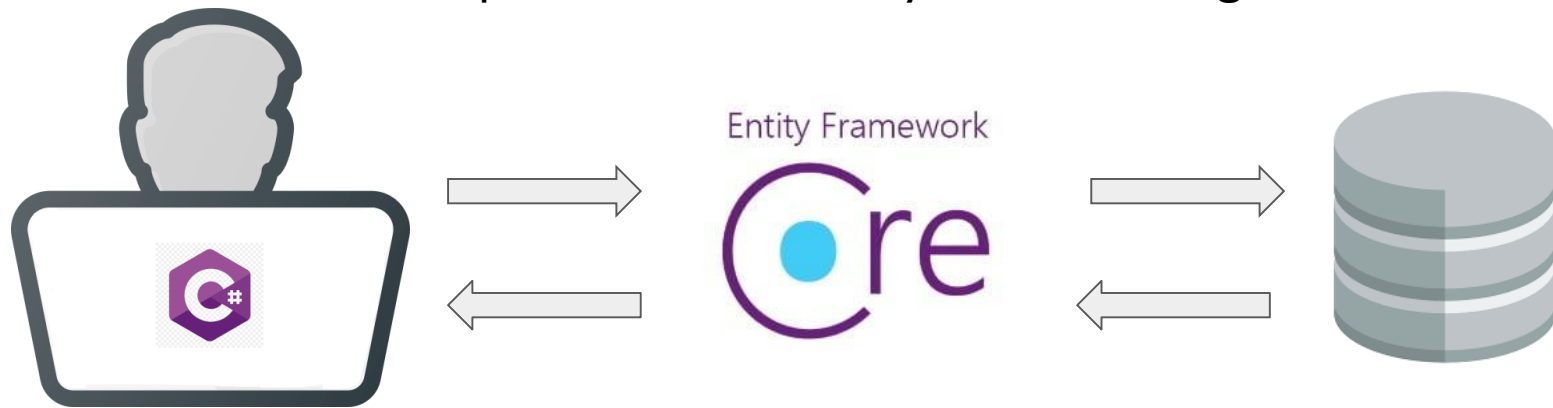


- **DB-First** : Génère le modèle objet en partant d'une base de données existante.
- **Code-First** : La base de données est créée sur base d'un modèle objet côté applicatif en suivant quelques règles.

Nous verrons dans ce cours l'approche Code-First

## 2. Comment ça marche ?

Le développeur ne voit et n'interagit réellement qu'avec EFCore. La DB est masquée par EFCore et nous n'avons plus de réel besoin d'y mettre les doigts.



EFCore se charge de la mise en place, de l'interrogation, et des mises à jours de la DB. Il gère également les connexions et leur statement. Le tout grâce au connecteur ADO

---

## 3. *Mise en place*

## 3.1 Les packages Nuget

Il est nécessaire d'installer la dépendance via nugets pour le fonctionnement de EF Core.

- **Microsoft.EntityFrameworkCore** => offre les classe nécessaire à l'utilisation d'EFCore
- **Microsoft.EntityFrameworkCore.SqlServer** => Permet l'utilisation avec Sql Server
- **Microsoft.EntityFrameworkCore.Tools** => Offre les commandes utilisables dans le PMC (dont nous parlerons plus tard)

*Remarque : le nuget SqlServer correspond au choix du SGDB sur laquelle nous travaillons*

## 3.2 Conventions, Fluent API et Data Annotations

Nous avons trois possibilités pour déclarer des contraintes, règles et associations au sein de nos classes.

- Par Conventions
- Par Data-Annotations
- Au travers de la Fluent-API

**Ces trois méthodes ne s'excluent pas l'une l'autre et peuvent s'avérer complémentaire ! Elles seront détaillées au travers de chaque concept.**

- Les Data-Annotations font partie des espaces de nom :  
***System.ComponentModel.DataAnnotations***  
***System.ComponentModel.DataAnnotations.Schema***
- La Fluent-API nécessite d'override la méthode *OnModelCreating()* de *DbContext* que nous verrons plus tard.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    Code de configuration Fluent-API
}
```

## 3.3 Fluent-API Best practices

Idéalement, nous créerons une classe de configuration par classe d'entité. (Pour peu que nous en ayons besoin).

Ces classes devront implémenter ***IEntityTypeConfiguration<T>*** . Le code Fluent-API sera alors écrit dans la méthode ***Configure(EntityTypeBuilder<T> builder)***. Nous devrons ensuite initialiser ces configurations dans la méthode **OnModelCreating()** de la classe Context (que nous verrons plus tard)

```
public class ContactConfig : IEntityTypeConfiguration<Contact>
{
    public void Configure(EntityTypeBuilder<Contact> builder)
    {
        builder.Property("LastName").IsRequired();
    }
}

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new ContactConfig());
}
```

---

## 3.4 Configuration des classes



## 3.4 Configuration des classes

Au moment de la migration vers la DB, chaque classe sera “convertie” en table, et chaque propriété en colonne de cette table. A moins qu’il ne soit fait mention du contraire

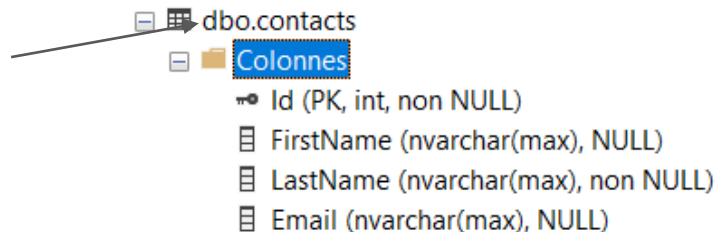
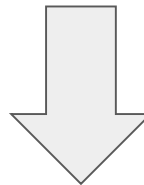
*(Excepté les propriétés de navigation que nous aborderons plus tard)*

Les types C# seront convertis en type DB par défaut si aucune mentions contraires n’est faites.

Par convention, le champ “**Id**” sera défini comme **Primary Key** du côté DB.

Un champ nommé **<Nom\_de\_la\_classe>Id** pourra également être prit pour **Primary Key** par défaut.

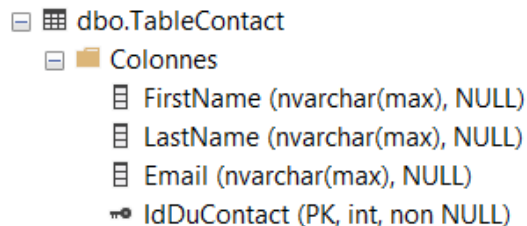
```
public class Contact
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
}
```



## 3.4 Configuration des classes

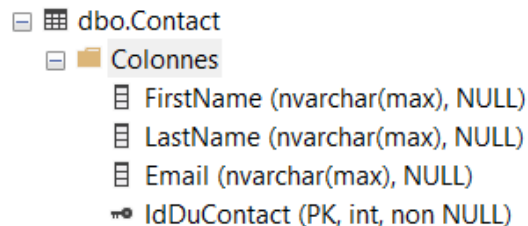
### Data-Annotations

```
[Table("TableContact")]
public class Contact
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int IdDuContact { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
}
```



### Fluent-API

```
modelBuilder.Entity<Contact>()
    .ToTable("Contact");
modelBuilder.Entity<Contact>()
    .HasKey("IdDuContact");
modelBuilder.Entity<Contact>()
    .Property(c => c.IdDuContact)
    .ValueGeneratedOnAdd();
```



## 3.4 Configuration des classes

### Primary key

Data-Annotations: **[Key]**

Fluent-API : Il faudra spécifier la clé directement à l'entité et non comme méthode d'une propriété particulière.

***modelBuilder.Entity<T>().HasKey("prop");***

### Auto Incrément

Par convention, EFCore ne nous permet pas de déclarer une colonne auto incrémentées.

Data-Annotations :  
**[DatabaseGenerated(DatabaseGeneratedOption.Identity)]**

Fluent-API :

**ValueGeneratedOnAdd();**

## 3.4 Configuration des classes

Remarquons que les colonnes *string/nvarchar* des tables créées en DB sont NULLABLE.

Par convention, la Nullabilité d'une colonne DB correspond à la nullabilité du type C#

Les types références (*string, object*) sont par défaut nullable.

Pour les rendre "Not Null", 2 options :

- **[Required]** en Data-annotations
- **.Required();** en Fluent-API

Pour les types structurels (ou valeur), il faudra utiliser **Nullable<T>** ou son raccourcis d'écriture **"?"** pour les rendre "NULL" en DB

## 3.4 Configuration des classes

Il est également possible d'intégrer des contraintes de Check à l'aide de la méthode

**HasCheckConstraint**("titre", "code SQL");

```
public void Configure(EntityTypeBuilder<Contact> builder)
{
    builder.HasCheckConstraint("CK_Email", "Email LIKE '__%@__%.%'");
}
```

La contrainte d'unicité devra être appliquée sur un index à l'aide des méthodes

**HasIndex**(champ\_a\_indexer).IsUnique();

```
public void Configure(EntityTypeBuilder<Contact> builder)
{
    builder.HasIndex(x => x.Email).IsUnique();
}
```

## 3.4 Configuration des classes (Conversion de type)

Type C#	Type SQL Server
INT	INT
Single	REAL
Double	FLOAT
String	NVARCHAR(MAX)
Decimal(x,x)	Decimal(x,x)
Bool	BIT
DateTime	DATETIME2(7)
Bytes[]	VARBINARY

## 3.4 Configuration des classes

Annotations	Utilité	Fluent-API
<b>[Required]</b>	Définit un champ "NOT NULL" en db	IsRequired();
<b>[Column("xxx", TypeName = "nvarchar(100)"]</b>	Fait correspondre la propriété au champ "xxx" de type "nvarchar(100)	HasColumnName("xxx"); HasColumnType("nvarchar(100)")
<b>[Table("xxx")]</b>	Fait correspondre la classe à la table "xxx"	ToTable("xxx");
<b>[StringLength(150, MinimumLength = 5)]</b>	Définit la longueur d'une chaîne de caractère avec un valeur minimum	/
<b>[MaxLength(xxx)]</b>	Définit la longueur maximale d'un string. Équivalent à <i>varchar(xxx)</i> en db	HasMaximumLength(xxx);
<b>[NotMapped]</b>	Classe ou propriété qui ne sera pas exporté vers la DB (ex : valeur calculée)	Ignore();
<b>[Key]</b>	Définit une propriété comme champ Primary Key	HasKey();

## 3.4 Configuration des classes

<b>[DatabaseGenerated()]</b>	Définit si une valeur est générée par la DB 3 options possibles	
<b>DatabaseGeneratedOption.Identity</b>	Définit qu'un champ sera rempli au premier enregistrement et ne sera plus pris en compte lors d'un <b>Update</b>	ValueGeneratedOnAdd();
<b>DatabaseGeneratedOption.Computed</b>	Valeur auto générée "OnAdd" et régénérée "OnUpdate" (au travers d'un trigger ou d'une valeur initialisée dans la classe)	ValueGeneratedOnAddOrUpdate();
<b>DatabaseGeneratedOption.None</b>	Signifie qu'on empêche la génération auto d'un champ (ex : une propriété Id de type INT qu'on ne voudrait pas utilisé comme PK)	ValueGeneratedNever();

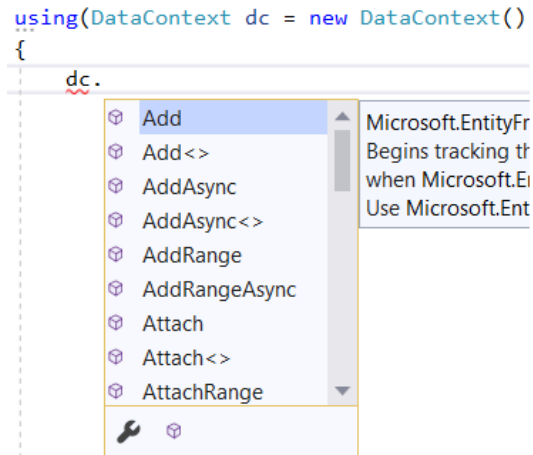


## 3.5 La classe DbContext

Une instance de **DbContext** représentera une session avec la base de données. Qui permettra d'interagir directement avec elle.

Nous devons créer une classe “*DataContext*” héritant de **DbContext** qui nous permettra d'utiliser l'objet *DataContext* pour toutes nos manipulations au sein de notre programme. **DbContext** nous fournit tous les outils nécessaires à l'utilisation de nos entités.

```
namespace DemoEFCore.Data
{
    public class DataContext : DbContext
    {
    }
}
```



## 3.6 La Connection string

Entity Framework repose sur les fondations d'**ADO.NET**, et pour toute connexion ADO nous avons besoin d'une *connection string*.

EF définit lui même la connexion à utiliser, se charge de gérer le `DbDataReader` et d'appeler les méthodes d'exécutions dépendantes d'ADO.

Dans notre classe héritant de **DbContext** nous allons override la méthode *OnConfiguring*.

Passons notre `connectionString` en tant qu'option nécessaire pour le constructeur.

La méthode d'extension *UseSqlServer* de `optionBuilder` signifie au système que nous utiliserons la factory SQL Server d'ADO.

```
public class DataContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            @"Data Source=DESKTOP-RGPQP6I\TFTIC2014;Initial Catalog=DemoEFCore;User ID=sa;Password=*****;"
        );
    }
}
```

## 3.7 DbSet<T>

Un **DbSet** est la collection C# correspondant aux tables de la DB. Il s'agit d'un type implémentant *IQueryable*, ce qui permet d'envoyer les requêtes LinQ directement vers le serveur. Contrairement à *IEnumerable* dont elle hérite et qui exécute LinQ côté client. Ce qui entraîne un gain de performance.

Chaque propriété de type **DbSet** prendra un type T.

Les **DbSet** pourront être directement accessibles au travers de notre objet de type *DbContext* en employant des requêtes LinQ.

```
public class DataContext : DbContext
{
    DbSet<Contact> contacts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("connectionString");
    }
}
```

## 3.7 DbSet<T>

Par convention, le nom de l'objet **DbSet<T>** sera repris comme nom de table

Data-Annotations

`[Table("Nom_Table")]`

Permet de définir le nom que prendra ma table en DB.

Fluent-API

Nous utiliserons la méthode

**ToTable(string);**

---

## 3.8 Héritage

## 3.8 Héritage

Il existe en réalité trois méthodes de mise en place d'un héritage

- **TPT (Table Per Type)** : Il y a une table partagée qui possède les propriétés de la classe de base et une table pour chaque classe héritée avec les propriétés qui lui sont spécifiques.
- **TPH (Table Per Hierarchy)** : Ici il n'y a qu'une seule table et un discriminant pour savoir sur quel type on se trouve
- **TPC (Table Per Concrete Class)** : Chaque classe héritée possède sa table avec ses propriétés qui lui sont propres plus les propriétés de la classe de base.

Entity Framework Core n'implémente actuellement que la méthode TPH (Affaire à suivre)

## 3.8 Héritage

```
public class Personne
{
    public int Id { get; set; }
}
public class Eleve : Personne
{
    public int NoteMoyenne { get; set; }
}
public class Professeur : Personne
{
    public string MatierePrincipale { get; set; }
}
```

Partant de ces 3 classes nous avons deux options pour mettre en place notre héritage.

EFCore va rassembler tous les champs dans une même table en y incluant un champ “Discriminator” qui permettra de distinguer le type d’enregistrement.

## 3.8 Héritage

### Conventions

```
public class DataContext : DbContext
{
    public DbSet<Personne> listePersonne { get; set; }
    public DbSet<Eleve> listeEleve { get; set; }
    public DbSet<Professeur> listeProfesseur { get; set; }
```

Nous définissons simplement nos DbSet<T>. EF conclura ce qu'il doit faire en voyant que plusieurs classes héritent d'une même classe mère

### Fluent-API

L'utilisation de la Fluent-API nous permet de définir comment nous souhaitons que le champ Discriminator soit configuré (un flag plutôt qu'un string par exemple)

```
public DbSet<Personne> listePersonne { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Personne>()
        .HasDiscriminator<string>("Type_personne")
        .HasValue<Eleve>("eleve")
        .HasValue<Professeur>("prof");
}
```



---

## 4. Les commandes

## 4. Les commandes


Pour générer la base de données, il sera nécessaire d'utiliser des lignes de commandes. Au travers de la console "Package manager console", il est possible d'utiliser les commandes venant de Nuget Package.

*Pour accéder à cette console :*

*Outils > Gestionnaire de package Nuget > Console de gestionnaire de package*

*(EN : Tools > Nuget package manager > Package Manager Console)*

Console du Gestionnaire de package

Source de packages : Tout  Projet par défaut : DemoEFCore

Tapez 'get-help NuGet' pour afficher toutes les commandes NuGet disponibles.

PM>

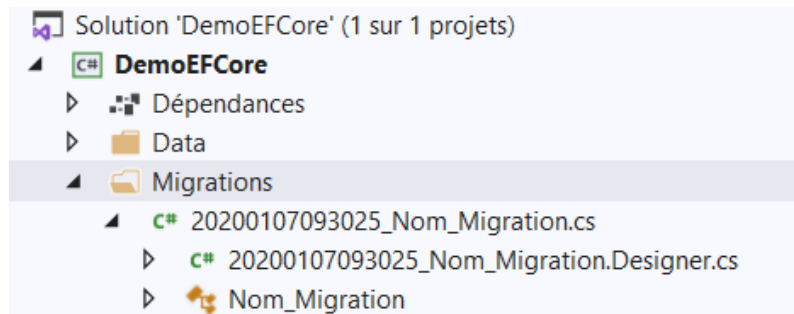
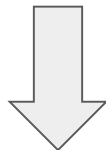
- ***get-help EntityFrameworkCore*** : Retourne toute les infos sur EFCore ainsi que la liste des commandes disponibles

## 4. Migration

**Syntaxe :** “*add-migration  
nom\_migration*”

Génère du code C# qui sera ensuite interprété par Entity Framework pour la création de la base de donnée. Ce code sera placé dans une classe nommée “*datetime\_nomMigration.cs*” dans un répertoire “Migrations” également créé par EFcore.

```
PM> Add-Migration Nom_Migration  
Build started...  
Build succeeded.
```



Cette classe hérite de *Migration* et override ses méthodes “*up*”, “*down*” et “*BuildTargetModel*”.

## 4. Migration

Nous créerons une nouvelle migration pour chaque mise à jour de la structure de notre DB. Seule la dernière migration en date sera prise en charge par la commande :

***Update-Database.*** (C'est la méthode Up() qui sera appelée)

Il est néanmoins possible de revenir à une migration précédentes en utilisant toujours

***Update-Database nom\_migration\_anterieur***

pour peu qu'une migration plus récente ait été appliquée. La méthode ***down()*** est alors appelée.

## 4. Migration

Chaque migration exécutée donnera lieu à un enregistrement dans une table DB créée et gérée par Entity Framework.

📊	dbo.__EFMigrationsHistory
📁	Colonnes
🔑	MigrationId (PK, nvarchar(150), non NULL)
📄	ProductVersion (nvarchar(32), non NULL)

Les scripts générés par la migration sont bien entendu modifiables avant exécution par la commande *Update-Database*.

```
public partial class seed : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "contacts",
            MiseEnPage
        )
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "contacts");
    }
}
```

## 4. Les commandes

- **Script-Migration** : Permet de voir le code SQL généré pour la migration vers la base de donnée

Attention : Le script est juste affiché !

Pas ajouté au projet !

```
1 IF OBJECT_ID(N'[__EFMigrationsHistory]') IS NULL
2 BEGIN
3     CREATE TABLE [__EFMigrationsHistory] (
4         [MigrationId] nvarchar(150) NOT NULL,
5         [ProductVersion] nvarchar(32) NOT NULL,
6         CONSTRAINT [PK__EFMigrationsHistory] PRIMARY KEY ([MigrationId])
7     );
8 END;
9
10 GO
11
12 CREATE TABLE [contacts] (
13     [ContactId] int NOT NULL IDENTITY,
14     [FirstName] nvarchar(max) NULL,
15     [LastName] nvarchar(max) NULL,
16     [Email] nvarchar(max) NULL,
17     CONSTRAINT [PK_contacts] PRIMARY KEY ([ContactId])
18 );
```

## 4. Les commandes

- **Remove-Migration** : Supprime la dernière migration créée à l'aide de "Add-migration" mais ne provoque pas de rollback de la DB.
- **Update-Database** : Exécute le code de la dernière migration généré par "Add-Migration"
- **Drop-Database** : Supprime la base de données renseignée dans la connection string du DbContext

---

# Exercise



# Exercice

- 1) Créer une classe Film. Chaque film sera défini par son titre (qui devra être unique), son année de sortie, son réalisateur, son acteur principal et son genre.

Aucun des champs ne peut être nullable, et d'une taille max de 100 pour les string.

Chaque film aura également un ID auto incrémenté et devra être sorti après 1975.

- 1) Déployer cette classe au niveau de votre DB en utilisant Entity Framework

- 1) Vérifier le résultat en DB.

---

## 5. Seed Data

## 5. Seed Data

Il s'agit d'une manière d'injecter des données directement dans la DB lors d'une migration. En passant par la Fluent-API. Il suffit d'utiliser la méthode `HasData(object T)`.

```
public class ContactConfig : IEntityTypeConfiguration<Contact>
{
    public void Configure(EntityTypeBuilder<Contact> builder)
    {
        builder.HasData(new Contact
        {
            ContactId = 1, FirstName = "Ator",
            LastName = "Termin", Email = "termin.ator@skynet.com"
        },
        new Contact
        {
            ContactId = 2, FirstName = "Chuck",
            LastName = "Norris", Email = "chuck.norris@demigod.com"
        });
    }
}
```

---

## 6. Utilisation du DbContext

## 6. Utilisation du DbContext

La plupart des manipulation que nous faisons sur des données font partie de l'ensemble CRUD (Create, Read, Update, Delete).

Entity Framework facilite grandement l'utilisation d'un CRUD grâce à quelques petites méthodes et à l'utilisation de LinQ.

## 6.1 Create

Il est tout d'abord nécessaire d'instancier un objet héritant de DbContext.

```
using (DataContext dc = new DataContext())
{
    dc.contacts.Add(new Contact
    {
        FirstName = "Steve",
        LastName = "Lorent",
        Email = "steve.lorent@bstrom.be"
    });

    dc.SaveChanges();
}
```

La méthode **Add**(*object T*) permet d'ajouter un objet au DbSet<T>

La méthode **SaveChanges()** de DbContext mettra à jour la DB avec le contenu du DbSet<T>

## 6.2 Read

Pour parcourir les données de DbContext, nous utiliserons **LinQ**.

Après instanciation nous pourrons accéder au “*DbSet*” membres de *DbContext*

```
using (DataContext dc = new DataContext())
{
    var c = dc.contacts.Where(x => x.Id == 1);
    foreach (Contact item in c)
    {
        Console.WriteLine("id :"+item.Id);
        Console.WriteLine("prenom : "+item.FirstName);
        Console.WriteLine("nom : "+item.LastName);
        Console.WriteLine("email : "+item.Email);
    }
}
```

## 6.3 Update

Il s'agit de la mise en application des deux points précédents. L'utilisation de LINQ et de la méthode `SaveChanges()`;

```
using (DataContext dc = new DataContext())
{
    var c = dc.contacts.Where(x => x.Id == 1).FirstOrDefault();

    if(c is Contact)
    {
        c.Email = "test@update.com";
        dc.SaveChanges();
    }
}
```



## 6.4 Delete

Nous utiliserons encore une fois **LinQ** pour récupérer notre objet et la méthode *Remove()* pour le supprimer.

```
using (DataContext dc = new DataContext())
{
    var c = dc.contacts.Where(x => x.Id == 1).FirstOrDefault();

    if (c is Contact)
    {
        dc.Remove(c);
    }
}
```

## 6.5 Exceptions

Les méthodes *Remove()* et *SaveChanges()* sont susceptibles de lever des exceptions en cas de problèmes.

Il existe une classe d'exception qui traite ces problèmes

### DbUpdateException

```
try
{
    dc.SaveChanges();
}
catch(DbUpdateException e)
{
    Console.WriteLine(e.Message);
}
```

---

# Exercices

# Exercices

1. Récupérer le fichier de configuration DataSet.cs sur le partage et l'ajouter au projet.  
(Adapter la classe à votre projet !!!)
2. Mettre à jour la base de données
3. Ajouter le film “Pacific Rim” (Acteur principal : Charlie Hunnam, Réal : Guillermo Del Toro, Année : 2013, Genre : Action)
4. Sélectionner et afficher tous les films sortis avant 2001
5. Mettre à jour tous les films Star Wars pour remplacer l'acteur principal par Harrison Ford
6. Supprimer tous les films de Charlie Hunnam

---

# 7. Les relations

---

# 7.1 One-To-One

## 7.1 One-to-One

Déclarée par convention en établissant des propriétés de navigation au sein des classes à mettre en relations.

📊	dbo.contacts
📁	Colonnes
📄	FirstName (nvarchar(max), NULL)
📄	LastName (nvarchar(max), NULL)
📄	Email (nvarchar(max), NULL)
🔗	DetailsId (FK, int, NULL)
🔑	ContactId (PK, int, non NULL)

📊	dbo.ContactDetail
📁	Colonnes
🔑	Id (PK, int, non NULL)
📄	BirthDate (datetime2(7), NULL)
🔗	IdContact (int, non NULL)
📄	AdditionnalInformation (nvarchar(max), NULL)

```
public class Contact
{
    public int ContactId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }

    public ContactDetail Details { get; set; }
}

public class ContactDetail
{
    public int Id { get; set; }
    public DateTime? BirthDate { get; set; }
    public string AdditionnalInformation { get; set; }

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

---

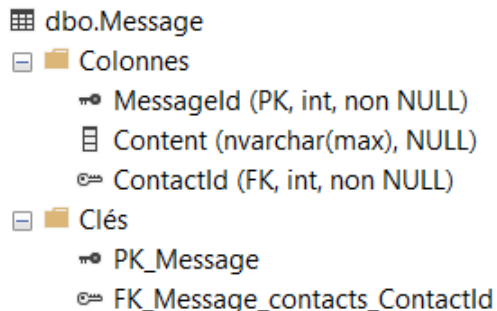
## 7.2 Many-To-One



## 7.2 Many-to-One

Il est possible par convention de définir une Many-to-One en ajoutant des propriétés de navigation dans les classes et sans obligation de définir un `DbSet<T>` du type dépendant.

En effet, EFCore se charge de suivre les propriétés de navigation et de définir les tables nécessaires au bon fonctionnement des relations mises en place.



```
public class Contact
{
    public int ContactId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }

    public List<Message> Messages { get; set; }
}
```

```
public class Message
{
    public int MessageId { get; set; }
    public string Content { get; set; }

    public int ContactId { get; set; }
    public Contact Sender { get; set; }
}
```

## 7.2 Many-to-One

Par Data-Annotations, nous pouvons spécifier une ***[ForeignKey(nameof(TypeParent))]*** sans avoir à spécifier un objet de type Parent dans la classe “enfant”. Le résultat est identique à la conventions.

Le seul avantage est de pouvoir déclarer des Foreign key composite.  
***[ForeignKey(“Property1”, “Property2”)]***

```
public class Contact
{
    public int ContactId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }

    public List<Message> Messages { get; set; }
}
```

```
public class Message
{
    public int MessageId { get; set; }
    public string Content { get; set; }

    [ForeignKey(nameof(Contact))]
    public int ContactId { get; set; }
}
```

## 7.2 Many-to-one

Nous choisirons d'utiliser la Fluent-API pour déclarer nos relations pour garder nos classes d'entités le plus clair possible.

```
modelBuilder.Entity<Contact>()  
    .HasMany(m => m.Messages)  
    .WithOne(s => s.Sender);
```

L'avantage reste de pouvoir déclarer des  
Foreign key composite.

*.HasForeignKey(c => new {"prop1", "prop2"});*

```
public class Message  
{  
    public int MessageId { get; set; }  
    public string Content { get; set; }  
    public Contact Sender { get; set; }  
}  
  
public class Contact  
{  
    public int ContactId { get; set; }  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
    public string Email { get; set; }  
    public List<Message> Messages { get; set; }  
}
```

---

## 7.4 Many-To-Many

## 7.3 Many-to-Many

Nous devons mettre en place une classe intermédiaire (à l'instar de SQL avec une N-N)

```
public class Meeting
{
    public int MeetingId { get; set; }
    public DateTime DateMeeting { get; set; }
    public string Location { get; set; }

    public List<ContactMeeting> ContactsMt { get; set; }
}
```

```
public class ContactMeeting
{
    public int ContactId { get; set; }
    public Contact Contact { get; set; }
    public int MeetingId { get; set; }
    public Meeting Meeting { get; set; }
}
```

```
public class Contact
{
    public int ContactId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }

    public List<Message> Messages { get; set; }
    public List<ContactMeeting> Meetings { get; set; }
}
```

Nous mettons en place les Foreign Key par conventions entre nos trois classes.

## 7.3 Many-to-Many

La deuxième étape consiste à configurer la classe intermédiaire dans la Fluent-API.

```
public class MeetingContactCFG : IEntityTypeConfiguration<ContactMeeting>
{
    public void Configure(EntityTypeBuilder<ContactMeeting> builder)
    {
        builder.HasKey(s => new { s.ContactId, s.MeetingId });

        builder.HasOne(mc => mc.Meeting)
            .WithMany(m => m.ContactsMt)
            .HasForeignKey(fk =>fk.MeetingId);

        builder.HasOne(cm => cm.Contact)
            .WithMany(c => c.Meetings)
            .HasForeignKey(fk => fk.ContactId);
    }
}
```

## 7.3 Many-to-Many

La dernière étape consiste à appeler cette configuration au niveau de la méthode ***OnModelCreating()*** de la classe *DbContext*

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new MeetingContactCFG());
}
```

Il ne reste qu'à générer la migration et mettre à jour la DB.

---

## 7.4 On Delete



## 7.4 OnDelete

Nous pourrions spécifier le comportement OnDelete sur les relations (comme en DB).

Ajoutons simplement la méthode ***OnDelete(DeleteBehavior.xxx)*** lors de la déclaration des Foreign Key.

```
builder.HasOne(cm => cm.Contact)
    .WithMany(c => c.Meetings)
    .HasForeignKey(fk => fk.ContactId).OnDelete(DeleteBehavior.Cascade);
```

## 7.4 OnDelete

**DeleteBehavior** est une Enumération contenant 4 choix.

<b>Cascade</b>	L'entité dépendante est supprimée avec l'entité principale
<b>SetNull</b>	L'entité dépendantes n'est pas supprimée mais ses foreign key sont définies à null
<b>ClientSetNull</b>	S'il y a des entité dépendantes, leur foreign key sont définies à null Si EFCore n'en trouve pas, les règles DB s'appliquent
<b>Restrict</b>	L'action Delete n'est pas appliquée aux entités dépendantes. L'entité principale ne peut être supprimée si elle a des entités dépendante

## 7.5 Données relationnelles

Il existe plusieurs manières de consommer les données en EFCore.

LinQ fait partie intégrante de cette consommation mais EFCore nous apporte son lot de méthode également.

Nous pouvons nous passer de créer des jointures en LinQ en utilisant les méthodes ***Include()*** et ***ThenInclude()***.

***Include("Propriété\_de\_navigation");***

Permet de peupler la propriété de navigation avec son contenu, puisque par défaut elle reste "NULL".

***ThenInclude("Propriété\_de\_navigation");***

Permet d'ajouter un niveau de navigation à ces propriétés (exemple : Many-to-Many qui va chercher ses information un niveau plus bas)

```
var FullContact = dc.contacts.Include(x => x.Meetings)
                             .ThenInclude(m => m.Meeting)
                             .ToList();
```

---

# Exercise

# Exercices

- 1) Mettre en place une classe d'entité Personne (id, nom, prenom). Ces personnes pourront être acteur et/ou réalisateur dans un film.

Chaque film n'aura qu'un seul réalisateur, mais plusieurs acteurs possibles.

- 1) Effectuer les changements nécessaires dans la classe Film et mettre à jour la base de données pour qu'elle puisse accueillir un nouveau jeu de données.
- 1) Afficher tous les films avec réalisateur et acteurs.

---

## 8. Références

## 8. Références

- Msdn Documentation Officielle Microsoft
  - <https://docs.microsoft.com/en-us/ef/core/>
- <https://www.learnentityframeworkcore.com/>
  
- C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development
  - Mark J. Price