

Rencontre 15

Application Web Database-First

Bases de données et programmation Web



- ❖ Génération d'un projet Web
 - ◆ BD, Modèles, contrôleurs, vues
- ❖ Interaction avec la BD
 - ◆ Tables, vues, procédures, déclencheurs

Précision : Beaucoup de notions dans ce cours sont présentes à des fins de **révision**. Si certaines notions vous semblent **familières**, c'est normal. C'est pour être sûr que tout le monde est **au même niveau** pour la suite du cours. Il y a tout de même beaucoup de **petites nouveautés** à travers les diapos, restez **attentifs** !

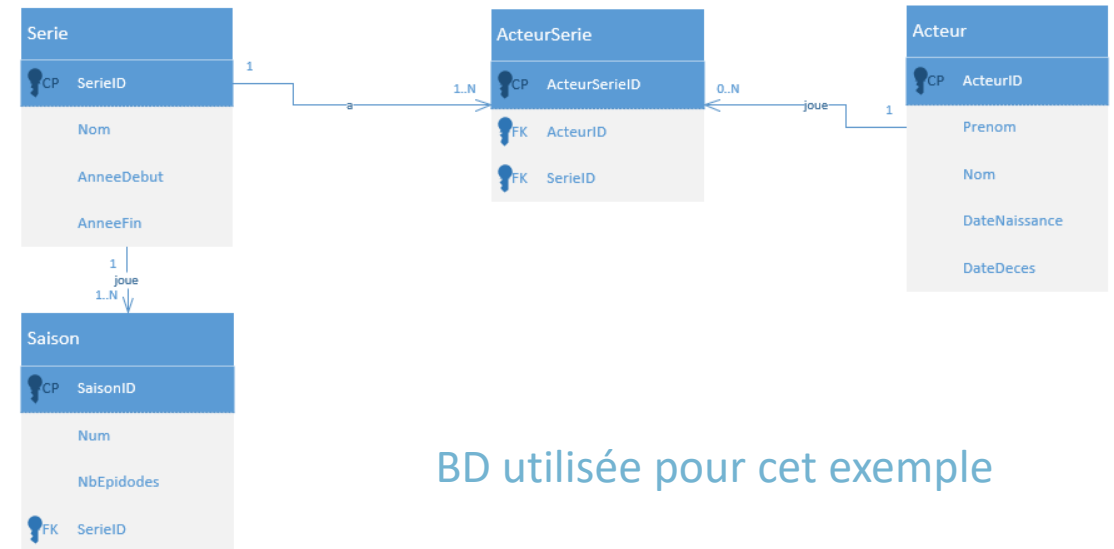


❖ Génération d'un projet Web DB-First

◆ Dans les prochaines diapositives, nous aborderons un exemple complet de génération d'un projet Web ASP.NET Core utilisant une base de données existante.

- Création du **projet** ASP.Net Core MVC
- **Connexion** à la base de données SQL Server **existante**
- Génération du **DbContext** et des **modèles**
- Génération des **contrôleurs**
- Générations des **vues**

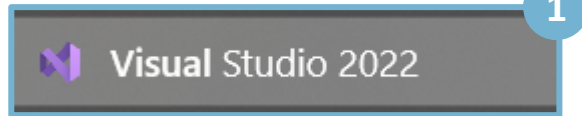
- Précision : Certaines étapes sont à prendre ou à laisser. (Notamment pour les contrôleurs et les vues)
- Même si vous êtes très à l'aise avec ASP.NET Core, **faites les étapes telles que présentées, particulièrement au niveau de la connexion à la BD.**



BD utilisée pour cet exemple

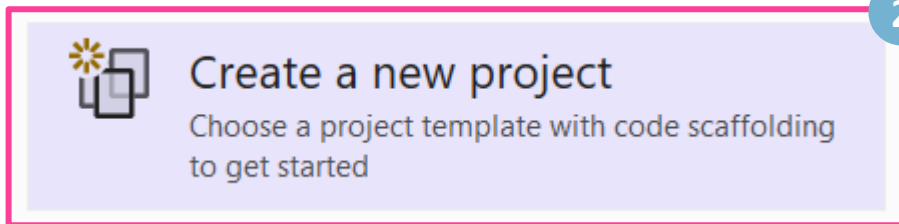


❖ Étape 1 : Création du projet

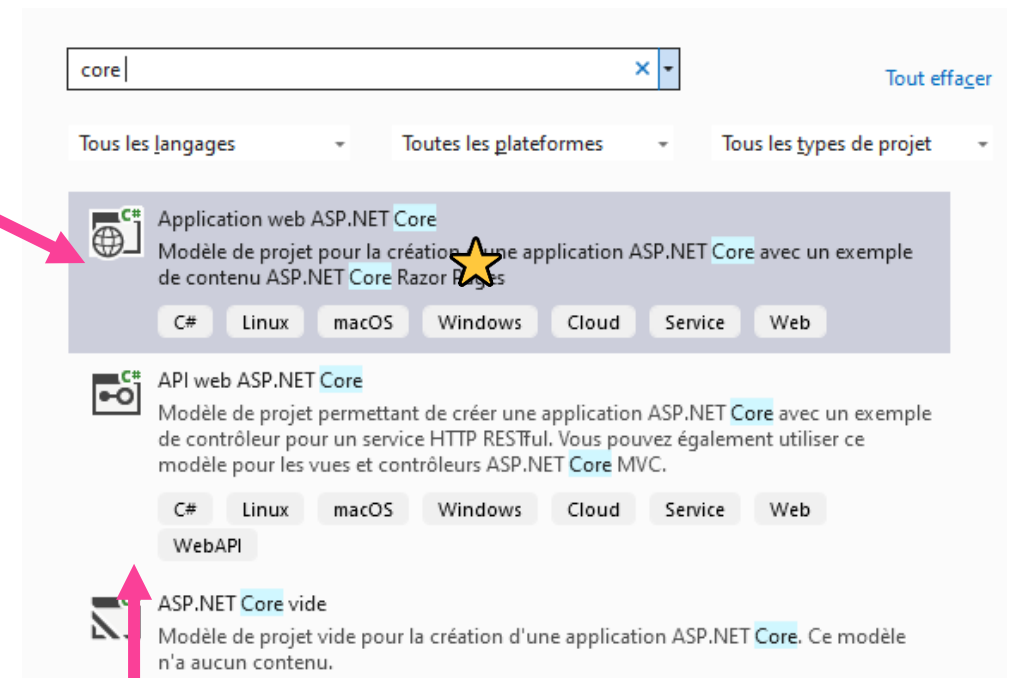


1

Nous utiliserons ce template dans ce cours



2



3

Ce template est utilisé en **Prog Web orientée services**, car aucune View Razor n'est nécessaire. (Angular s'occupe de l'interface utilisateur)



❖ Étape 1 : Création du projet

Configurer votre nouveau projet

Application web ASP.NET Core C# Linux macOS Windows Cloud Service Web

Nom du projet

Rencontre15

Emplacement

D:\Cegep\4D5\H24\R15 Projet Web BD First

Nom de la solution ⓘ

Rencontre15

☒ Placer la solution et le projet dans le même répertoire

Projet sera créé en tant que « D:\Cegep\4D5\H24\R15 Projet Web BD First\Rencontre15\ »

4

Informations supplémentaires

Application web ASP.NET Core C# Linux macOS Windows Cloud Service Web

Infrastructure ⓘ

.NET 8.0 (Prise en charge à long terme)

On prend .NET 8.0 !

Type d'authentification ⓘ

Aucun

☐ Configurer pour HTTPS ⓘ

☐ Activer la prise en charge du conteneur ⓘ

Conteneur OS ⓘ

Linux

Type de build du conteneur ⓘ

Dockerfile

☐ N'utilisez pas d'instructions de niveau supérieur. ⓘ

☐ Inscrire dans l'orchestration de .NET Aspire ⓘ

5

Ne pas configurer pour HTTPS. Ne pas cocher cette case!



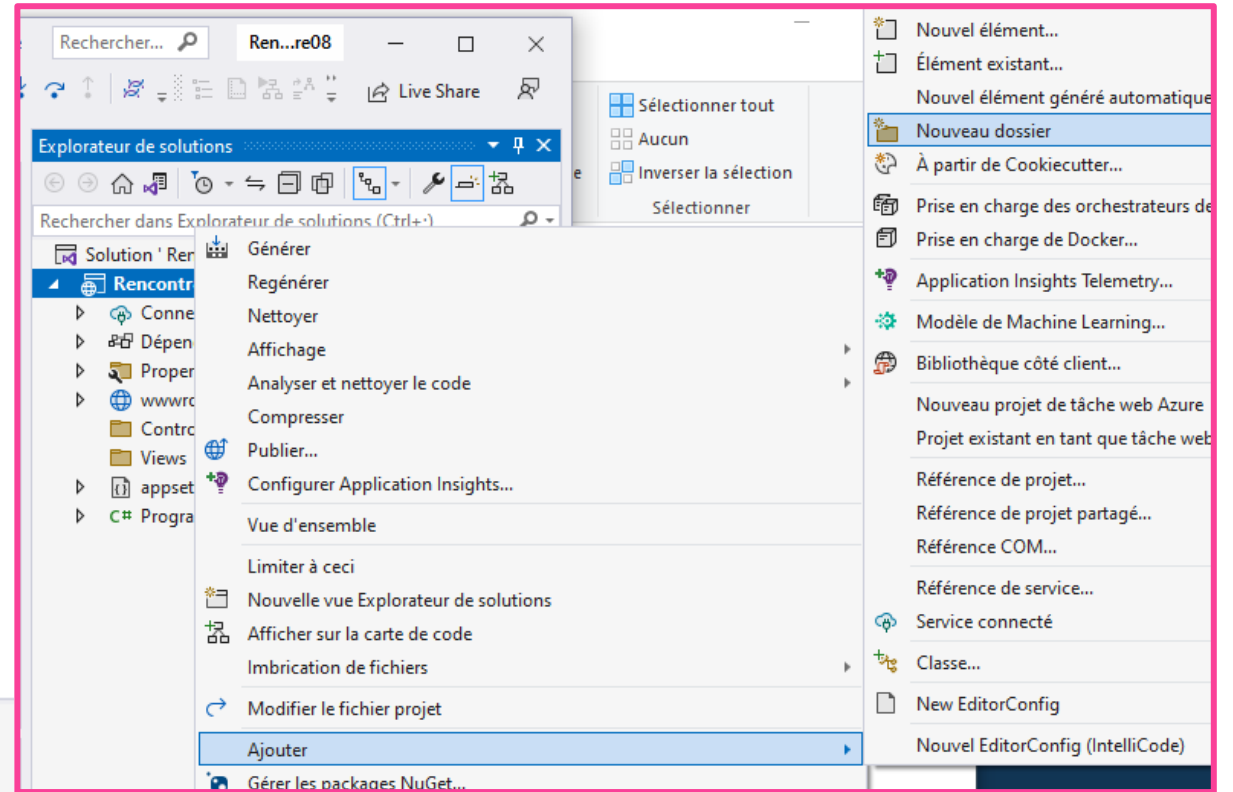
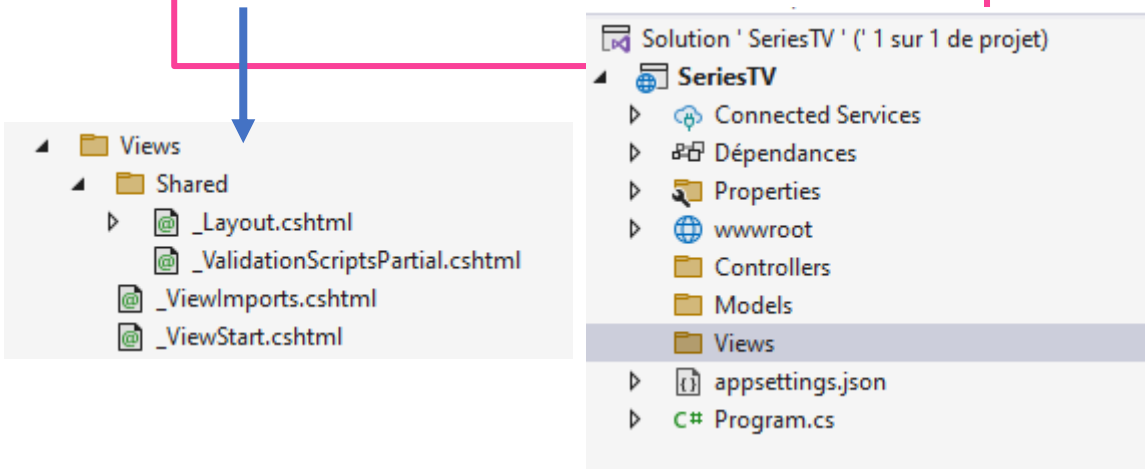
❖ Étape 1 : Création du projet

6

Créez les dossiers **Controllers**, **Models** et **Views**

(Le dossier **Data** sera généré automatiquement.)

Déplacez dans Views les éléments ci-dessous du dossier **Pages**, puis supprimez le dossier **Pages**





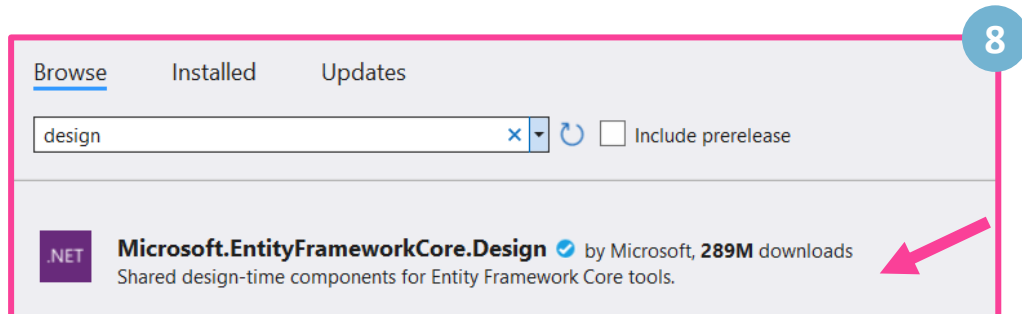
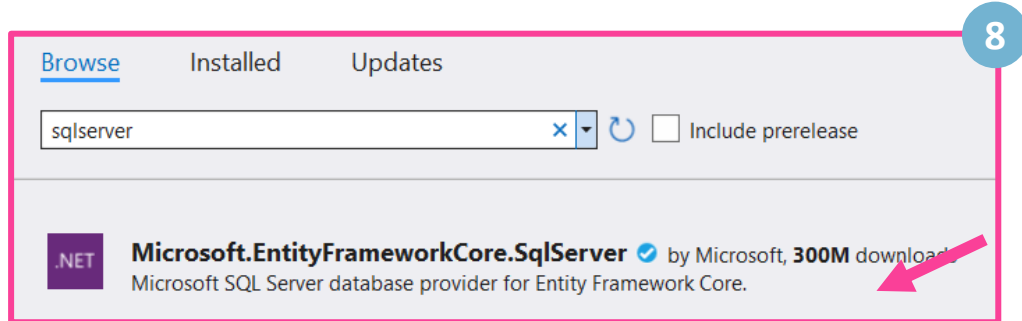
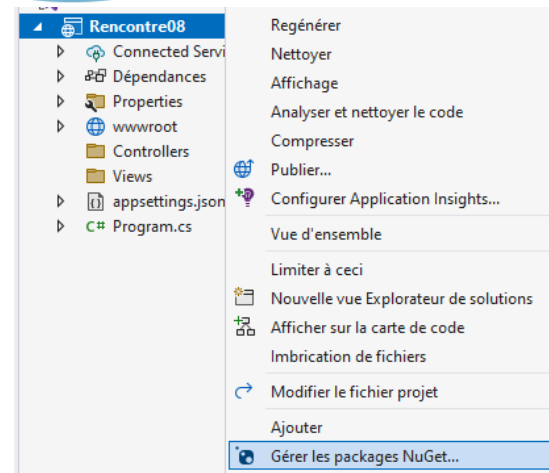
❖ Étape 1 : Création de la BD

- ◆ Nous avons besoin d'une BD existante.
- ◆ Donc ouvrez SSMS, connectez-vous et exécutez le script de création de la BD
Script BD SeriesTV.sql

Génération d'un projet



❖ Étape 1 : Création du projet







- Installez les packages **SqlServer** et **Design** d'**EntityFrameworkCore**. Nous utiliserons cet Object Relational Mapping (**ORM**) pour simplifier l'interaction entre la base de données et l'application Web.

- Utilisez la dernière **version 9.0..**  pour ces 2 packages.

Générez votre solution tout de suite.

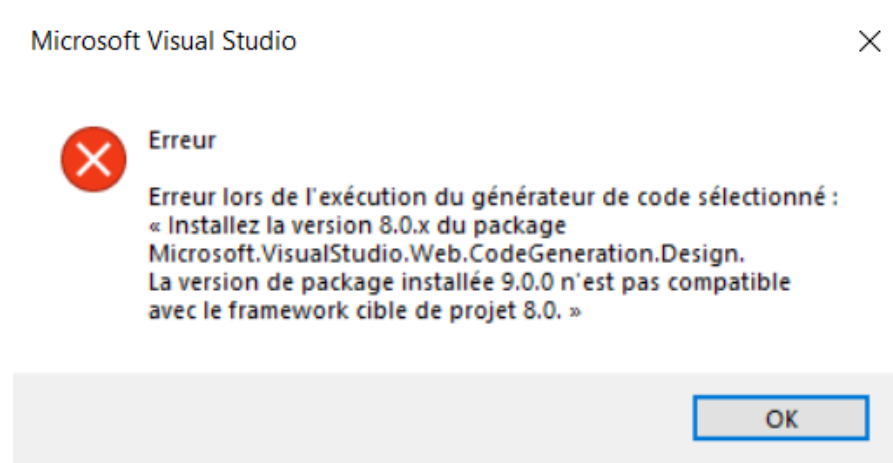


❖ Ajoutez 2 autres packages avec les versions ci-dessous:

	Microsoft.EntityFrameworkCore.Design par Microsoft Shared design-time components for Entity Framework Core tools.	9.0.2
	Microsoft.EntityFrameworkCore.SqlServer par Microsoft Microsoft SQL Server database provider for Entity Framework Core.	9.0.2
	Microsoft.EntityFrameworkCore.Tools par Microsoft Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.	8.0.13 9.0.2
	Microsoft.VisualStudio.Web.CodeGeneration.Design par Microsoft Code Generation tool for ASP.NET Core. Contains the dotnet-aspnet-codegenerator command used for generating controllers and views.	8.0.7 9.0.0

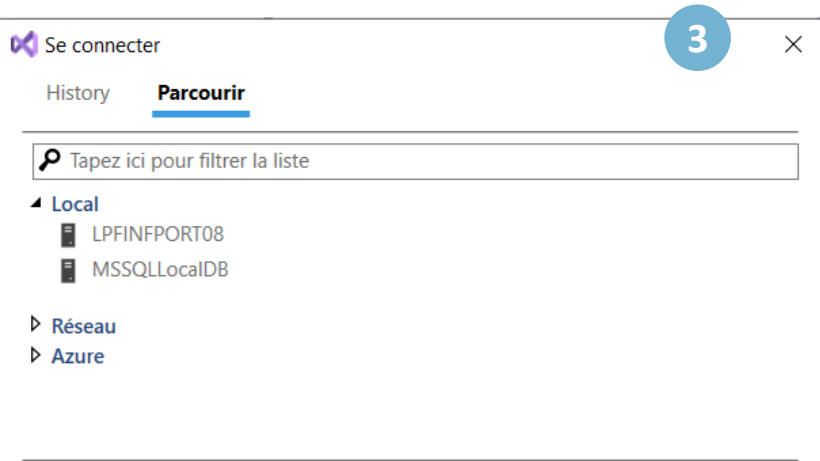
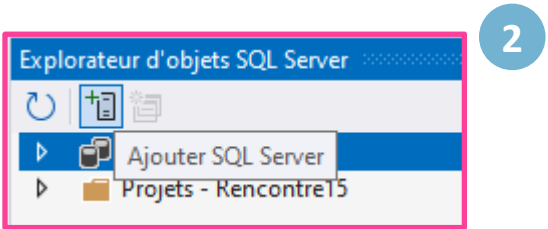
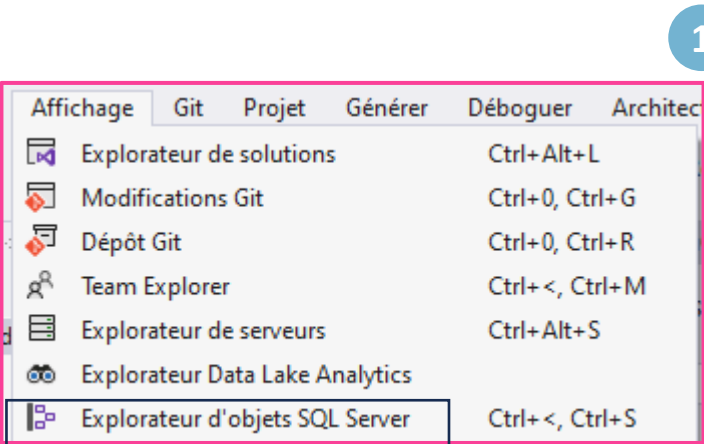


- ❖ Nous avons besoin d'installer des versions antérieures pour 2 packages car la génération automatique d'un contrôleur et de ses vues ne fonctionnent pas avec les versions les plus récentes.
- ❖ Si on essaie avec les versions les plus récentes de ces 2 packages on obtient le message d'erreur suivant:





❖ Étape 2 : Connexion à la base de données existante



Nom du serveur : LPFINFPORT08

Authentification : Authentification Windows

Nom d'utilisateur : LABORATOIRE\chantal.vallieres

Mot de passe :

☐ Se souvenir du mot de passe

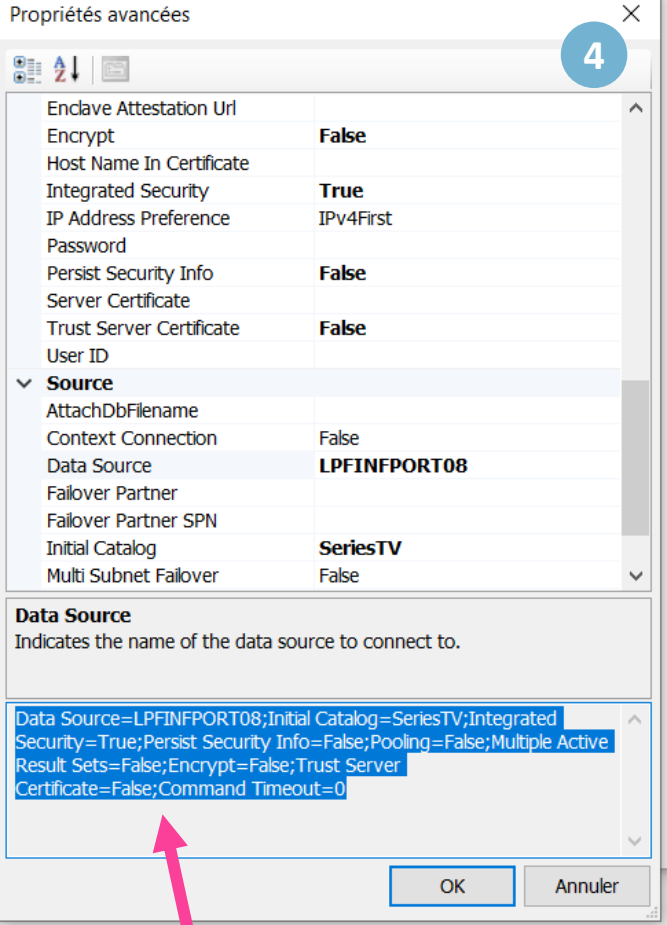
Nom de la base de données : SeriesTV

Crypter : Facultatif (Faux)

Certificat de serveur de confiance : False

Sélectionnez votre BD existante

Cliquez ici après avoir sélectionné votre nom de BD



Copiez cette chaîne de connexion dans votre presse-papier !



❖ Étape 2 : Connexion à la base de données existante

5

Connect

History **Browse**

Type here to filter the list

Local

- DESKTOP-3DDQ1QU\SQLEXPRESS
- DESKTOP-3DDQ1QU\SQLEXPRESS01
- DESKTOP-3DDQ1QU\SQLEXPRESS
- DESKTOP-3DDQ1QU\SQLEXPRESS01
- MSSQLLocalDB
- ProjectsV13

Network

Azure

Server Name:

Authentication:

User Name:

Password:

☐ Remember Password

Database Name:

Advanced...

Connect Cancel

- Finalement, ajoutez une petite section dans `appsettings.json` pour y intégrer votre **chaîne de connexion**. (Que vous aviez mise dans votre presse-papier)

```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft.AspNetCore": "Warning"  
    }  
  },  
  "AllowedHosts": "*",  
  "ConnectionStrings": {  
    "BDSeriesTV": "Data Source=LPINFPORT08;Initial Catalog=SeriesTV;Integrated Security=SSPI;"  
  }  
}
```

Remplacez **Data Source=LPINFPORT...** par **Data Source=.** pour que votre projet fonctionne si vous changez d'ordi...

Générez votre solution tout de suite.



❖ Étape 3 : Génération du DbContext et des Models

2

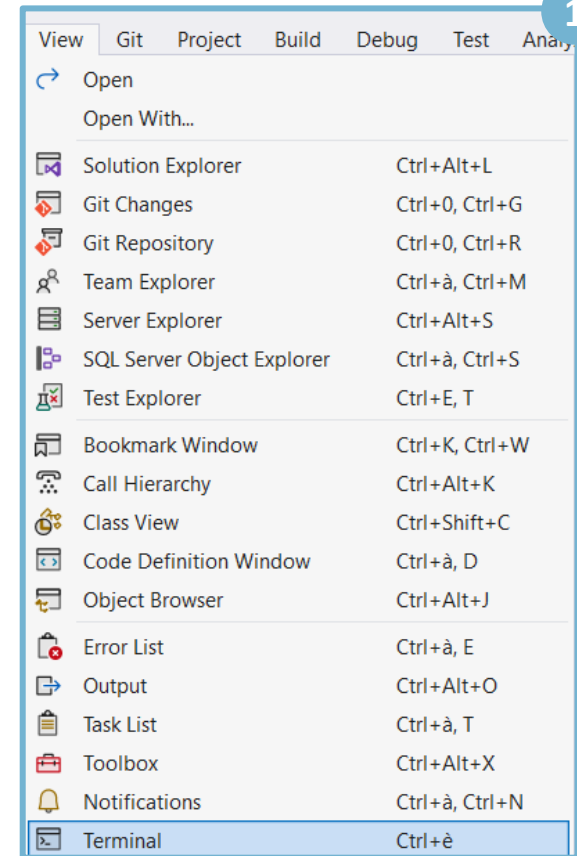
```
Developer PowerShell
+ Developer PowerShell
*****
** Visual Studio 2022 Developer PowerShell v17.4.4
** Copyright (c) 2022 Microsoft Corporation
*****
PS D:\base de données et prog web\notes_cours\web\Sem09>
```

- Si vous êtes à la maison, vous pourriez avoir besoin d'installer ou de mettre à jour les outils Entity Framework :

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

- Affichage/Terminal. Pour générer le DbContext et les Models, utilisez la commande suivante :

```
dotnet ef dbcontext scaffold Name=BDSeriesTV
Microsoft.EntityFrameworkCore.SqlServer -o Models --context-dir
Data --data-annotations
```



Remplacez **BDSeriesTV** par le nom de **votre** chaîne de connexion.



❖ Étape 3 : Génération du DbContext et des Models

◆ Précisions pour la commande

```
dotnet ef dbcontext scaffold Name=BDSeriesTV Microsoft.EntityFrameworkCore.SqlServer  
-o Models --context-dir Data --data-annotations
```

Les Models seront générés
dans le répertoire **Models**

Le DbContext sera généré
dans le répertoire **Data**

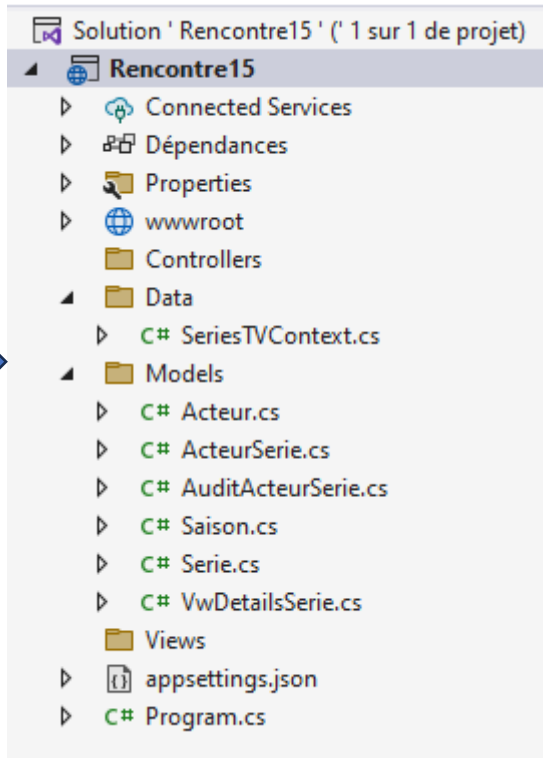
L'usage de **Data Annotations** sera priorisé (lorsque
possible) à la place d'instructions avec **Fluent API**.

```
[StringLength(100)]  
14 references  
public string Nom { get; set; } = null!;
```



❖ Étape 3 : Génération du DbContext et des Models

Après avoir généré le **DbContext**, il faudra le configurer dans **Program.cs**



```
// Add services to the container.  
builder.Services.AddRazorPages();  
  
builder.Services.AddDbContext<SeriesTvContext>(  
    options => options.UseSqlServer(builder.Configuration.GetConnectionString("BDSeriesTV")));  
  
var app = builder.Build();
```

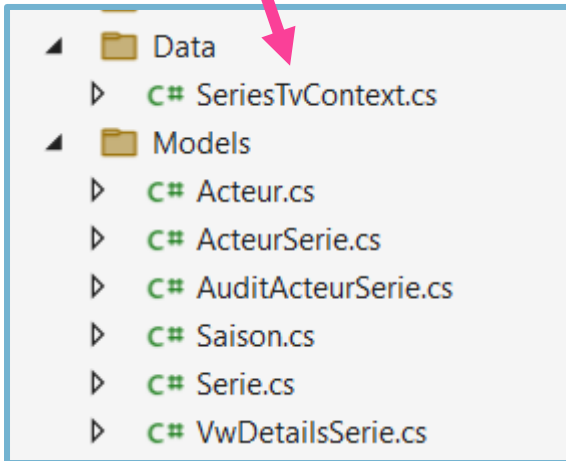


❖ Étape 3 : Génération du DbContext et des Models

◆ Résultat

Le DbContext contient des **DbSet<T>**. Les **DbSet<T>** sont une représentation de nos **tables** dans l'application Web.

DbContext dans le répertoire **Data**



Il y a également des instructions **Fluent API** pour décrire certaines **propriétés** de nos entités. Certaines contraintes ont plutôt la forme de **[DataAnnotations]** directement dans les classes des Models.

```
0 references
public virtual DbSet<Acteur> Acteurs { get; set; }

0 references
public virtual DbSet<ActeurSerie> ActeurSeries { get; set; }

0 references
public virtual DbSet<AuditActeurSerie> AuditActeurSeries { get; set; }

0 references
public virtual DbSet<Saison> Saisons { get; set; }
```

```
0 references
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Acteur>(entity =>
    {
        entity.HasKey(e => e.ActeurId).HasName("PK_Acteur_ActeurID");
    });

    modelBuilder.Entity<ActeurSerie>(entity =>
    {
        entity.HasKey(e => e.ActeurSerieId).HasName("PK_ActeurSerie_ActeurSerieID");

        entity.ToTable("ActeurSerie", "Series", tb => tb.HasTrigger("TR_ActeurSerie_idActeurSerie"));

        entity.HasOne(d => d.Acteur).WithMany(p => p.ActeurSeries).HasConstraintName("FK_ActeurSerie_ActeurID");

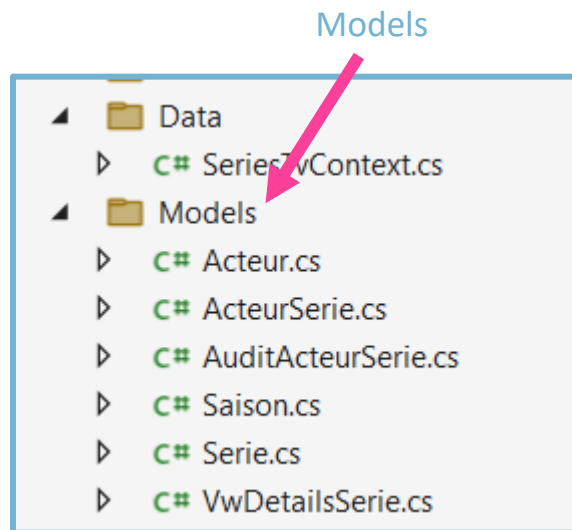
        entity.HasOne(d => d.Serie).WithMany(p => p.ActeurSeries).HasConstraintName("FK_ActeurSerie_SerieID");
    });

    modelBuilder.Entity<AuditActeurSerie>(entity =>
```




❖ Étape 3 : Génération du DbContext et des Models

◆ Résultat



```
[Table("Serie", Schema = "Series")]
5 references
public partial class Serie
{
    [Key]
    [Column("SerieID")]
    1 reference
    public int SerieId { get; set; }

    [StringLength(100)]
    0 references
    public string Nom { get; set; } = null!;

    0 references
    public int AnneeDebut { get; set; }

    0 references
    public int? AnneeFin { get; set; }

    [InverseProperty("Serie")]
    1 reference
    public virtual ICollection<ActeurSerie> ActeurSeries { get; } = new List<ActeurSerie>();

    [InverseProperty("Serie")]
    1 reference
    public virtual ICollection<AuditActeurSerie> AuditActeurSeries { get; } = new List<AuditActeurSerie>();

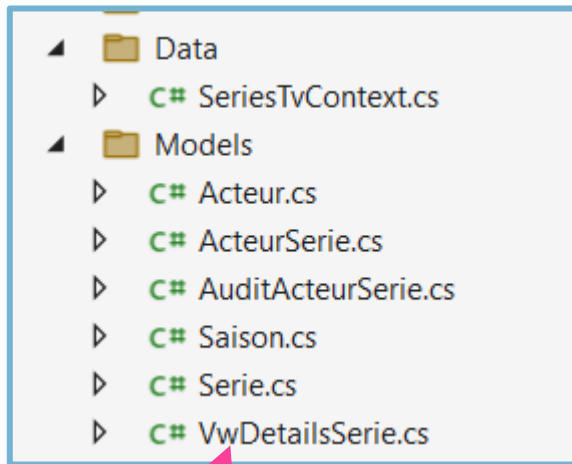
    [InverseProperty("Serie")]
    1 reference
    public virtual ICollection<Saison> Seasons { get; } = new List<Saison>();
}
```

Comme on a utilisé l'option **--data-annotations** dans la commande, on peut retrouver directement dans les classes des Models les contraintes qui peuvent être représentées par des **[DataAnnotations]**



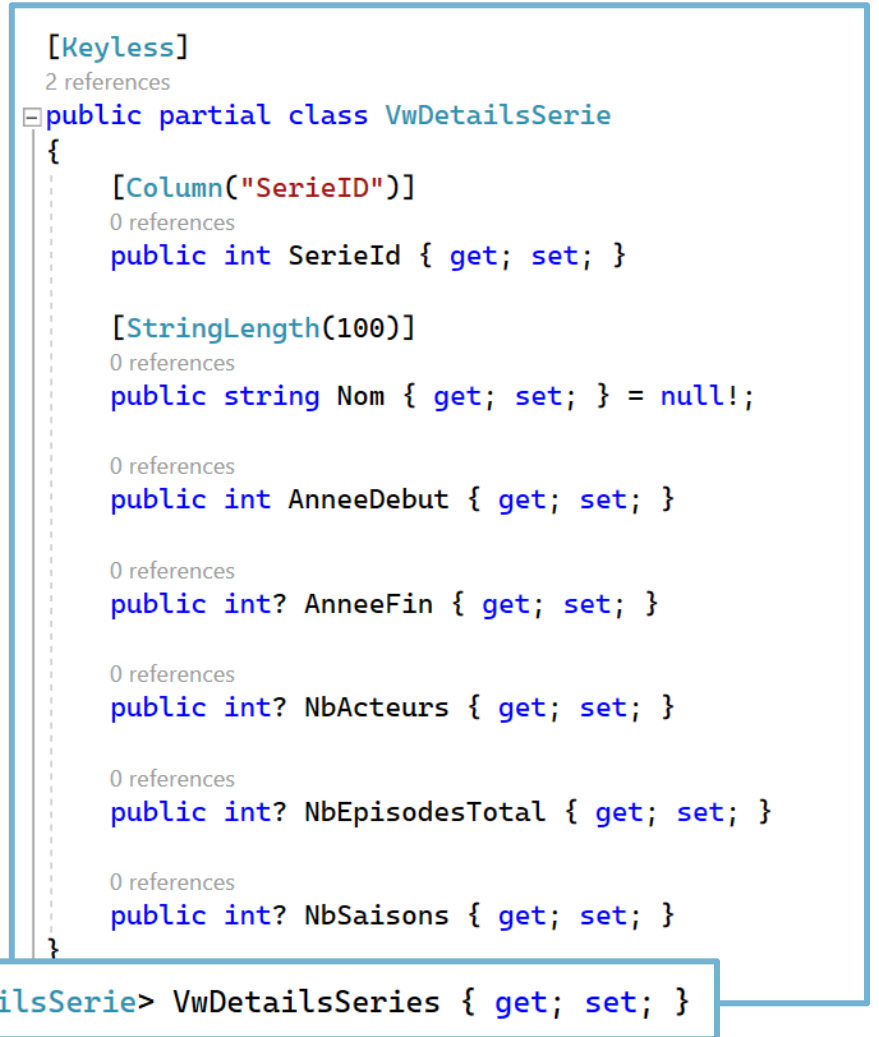
❖ Étape 3 : Génération du DbContext et des Models

◆ Résultat



On retrouve également nos vues !

Les vues ont un `DbSet<T>` dans le DbContext comme les tables

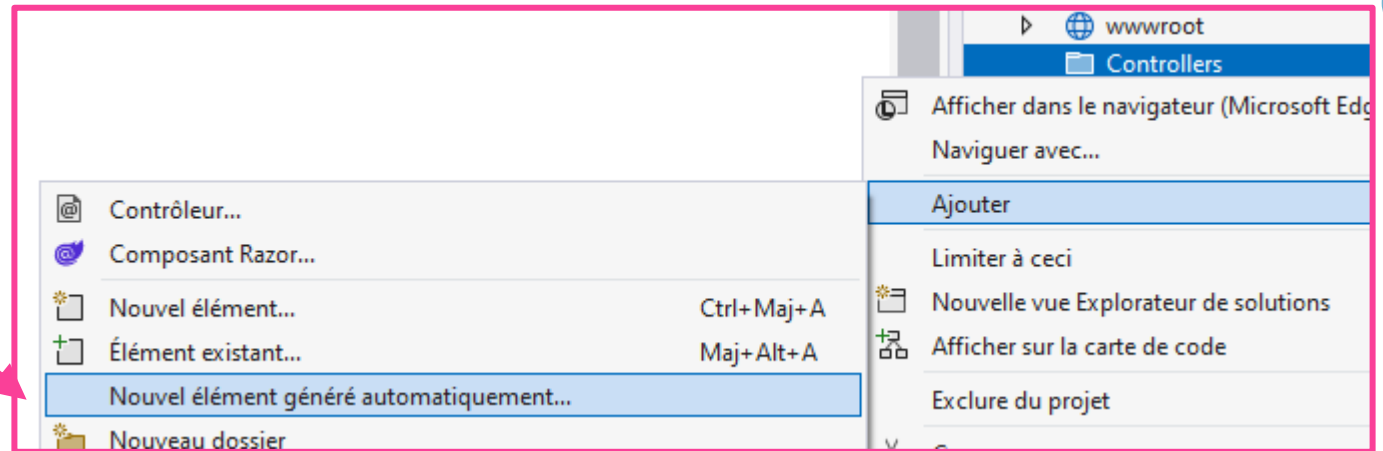




❖ Étape 4 : Génération des contrôleurs et des vues

- ◆ On peut aussi créer nos contrôleurs nous-mêmes. Les contrôleurs auto-générés nous donnent toutefois accès à des exemples fonctionnels pour manipuler et faire des requêtes sur une table dans la base de données.

**Générez votre solution tout de suite
AVANT d'auto-générer les contrôleurs.**



1

Ajouter un nouvel élément généré automatiquement

▲ Installé

▲ Éléments communs

API

Composant Razor

▶ MVC

Pages Razor

Identité

Mise en page



Zone MVC



Contrôleur MVC avec des actions de lecture/écriture



Contrôleur MVC avec vues, utilisant Entity Framework

2



❖ Étape 4 : Génération des contrôleurs et des vues

3

Ajouter Contrôleur MVC avec vues, utilisant Entity Framework

Classe de modèle: Serie (Rencontre15.Models)

Classe DbContext class: SeriesTVContext (Rencontre15.Data)

Vues

☒ Générer des vues

☒ Bibliothèques de scripts de référence

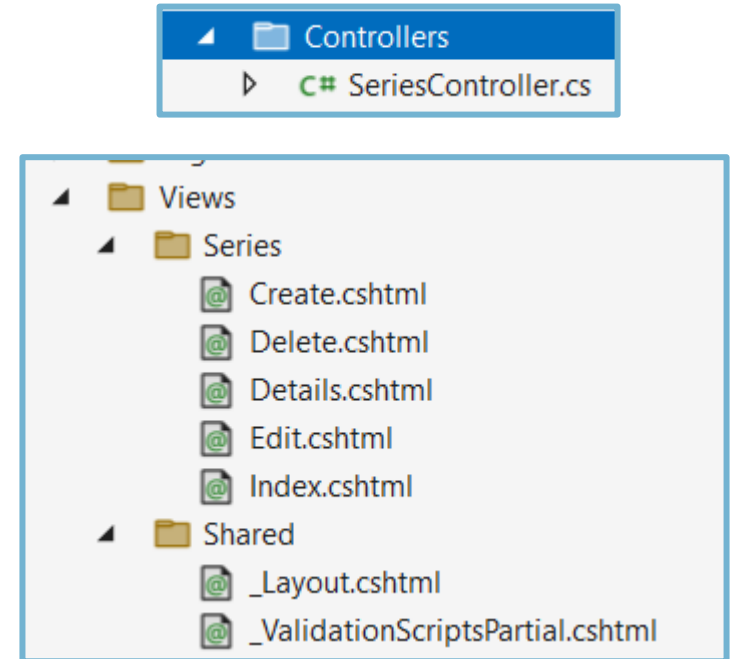
☒ Utiliser une page de disposition

~/Views/Shared/_Layout.cshtml

(Laissez vide s'il est défini dans un fichier Razor _viewstart)

Nom du contrôleur: SeriesController

Ajouter Annuler



- Cela nous génère donc un nouveau **contrôleur** ainsi que **5 vues**.
- Le **contrôleur** et les **vues** nous permettent de faire 5 opérations de base sur l'entité (le Model) choisie : **Voir toute** la table (Index.cshtml), **voir une** seule rangée (Details.cshtml), **créer** une rangée (Create.cshtml), **modifier** une rangée (Edit.cshtml) et **supprimer** une rangée (Delete.cshtml).



❖ Étape 5 : Routage de base pour accéder à nos vues

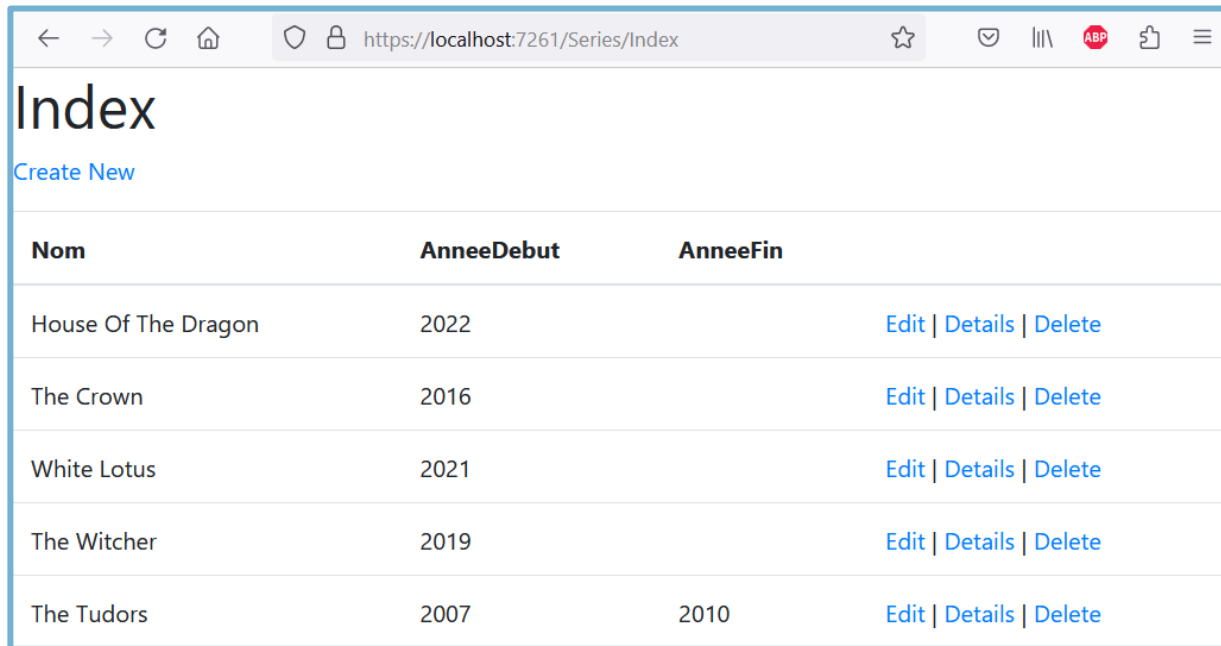
- ◆ Dans **Program.cs**, on ajoute un bloc de code pour que le routage fonctionne.
 - On en profite pour spécifier la page par défaut de notre choix.

```
app.UseAuthorization();  
  
app.MapControllerRoute(  
    name: "default",  
    pattern: "{controller=Series}/{action=Index}/{id?}"  
);  
  
app.MapRazorPages();
```



❖ Étape 6 : Tester le projet

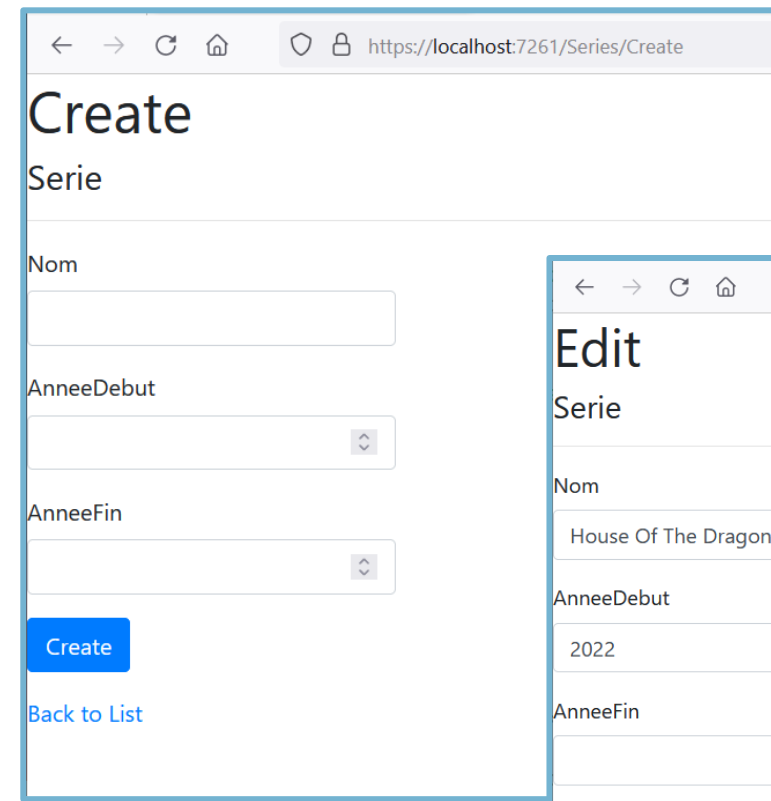
- ◆ On peut déjà faire des INSERT, des UPDATE et des DELETE dans nos tables via l'application.



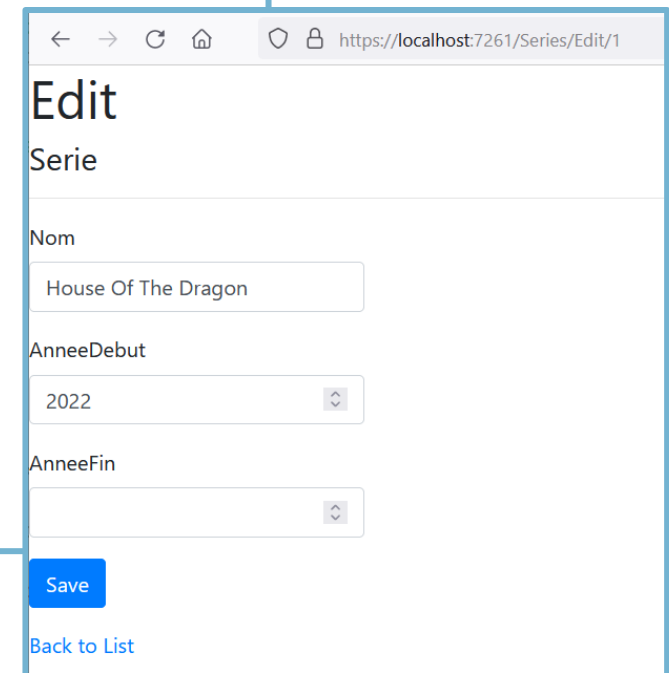
A screenshot of a web browser showing the 'Index' page at <https://localhost:7261/Series/Index>. The page has a 'Create New' link and a table with columns 'Nom', 'AnneeDebut', and 'AnneeFin'. The table contains five rows of data, each with 'Edit', 'Details', and 'Delete' links.

Nom	AnneeDebut	AnneeFin
House Of The Dragon	2022	
The Crown	2016	
White Lotus	2021	
The Witcher	2019	
The Tudors	2007	2010

- Bien entendu, ces vues ne correspondent pas forcément à ce dont on a besoin. Elles sont surtout une source d'exemples fonctionnels pour les opérations **CRUD**. (Create, Retrieve, Update, Delete)
- Rien ne vous empêche de faire vos propres vues ou de modifier celles auto-générées.



A screenshot of a web browser showing the 'Create' page at <https://localhost:7261/Series/Create>. The page has a title 'Create Serie' and form fields for 'Nom', 'AnneeDebut', and 'AnneeFin'. There is a 'Create' button and a 'Back to List' link.



A screenshot of a web browser showing the 'Edit' page at <https://localhost:7261/Series/Edit/1>. The page has a title 'Edit Serie' and form fields for 'Nom', 'AnneeDebut', and 'AnneeFin'. The 'Nom' field is pre-filled with 'House Of The Dragon'. There is a 'Save' button and a 'Back to List' link.



❖ Étape 7 : Services (ou « Repositories »)

- ◆ Les contrôleurs auto-générés interagissent directement avec le DbContext.
 - Généralement, on n'aime pas ça. On préfère introduire une petite *couche* entre les **contrôleurs** et le **DbContext** pour mieux encapsuler les opérations avec la base de données et retirer du code répétitif dans les contrôleurs.
 - **Repository Pattern + Unit of Work** est une architecture qui *pourrait* être la solution, mais c'est un peu de la *suringénierie** :
 - Cette architecture essaye de régler certains problèmes que DbContext gère déjà. (Comme la **concurrency**. DbContext est déjà un « Unit of Work » qui gère la concurrence)
 - La petite échelle de nos projets nous amène à utiliser des solutions plus économes.
 - Souvent, on utilise une solution plus modeste comme la couche Service. Vous verrez cela dans votre cours **Prog Web orientée services**.
 - NOUS, on va quand même utiliser l'interaction directe avec le DbContext.

*Suringénierie : Conception d'un produit avec de la surqualité, ce qui rajoute inutilement de la complexité et du coût.



❖ Interaction avec la BD

- ◆ DML (Insert, Update, Delete)
- ◆ Vues
- ◆ **Procédures**
- ◆ Déclencheurs



❖ Insert (Create), Update et Delete

◆ Exemple avec Create

1 - Dans les **vues auto-générées**, on a des exemples de formulaires / boutons avec de la **validation côté client** grâce aux **Tag Helpers**. Avec de la validation côté client, le formulaire n'est tout simplement pas envoyé au serveur si les données ne sont pas valides. *Cette validation peut être contournée par l'utilisateur cela dit !*

```
<form asp-action="Create">
  <div asp-validation-summary="ModelOnly" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="Nom" class="control-label"></label>
    <input asp-for="Nom" class="form-control" />
    <span asp-validation-for="Nom" class="text-danger"></span>
  </div>
  <div class="form-group">
    <label asp-for="AnneeDebut" class="control-label"></label>
    <input asp-for="AnneeDebut" class="form-control" />
    <span asp-validation-for="AnneeDebut" class="text-danger"></span>
  </div>
  <div class="form-group">
    <label asp-for="AnneeFin" class="control-label"></label>
    <input asp-for="AnneeFin" class="form-control" />
    <span asp-validation-for="AnneeFin" class="text-danger"></span>
  </div>
  <div class="form-group">
    <input type="submit" value="Create" class="btn btn-primary" />
  </div>
</form>
```

1

2 - Dans nos **contrôleurs**, on retrouve de la validation côté serveur : **ModelState.IsValid** se sert des [DataAnnotations] du Model. Si les données reçues sont invalides, on ne dérange même pas la BD ! Le [Bind(...)] permet de **whitelister**, autoriser seulement certaines des propriétés pour empêcher l'utilisateur d'envoyer des données superflues / malicieuses. L'action Create ajoute à la BD la nouvelle série si elle est valide ensuite.

```
public async Task<IActionResult> Create([Bind("SerieId,Nom,AnneeDebut,AnneeFin")] Serie serie)
{
    if (ModelState.IsValid)
    {
        _context.Series.Add(serie);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(serie);
}
```

2



❖ Insert (Create), Update et Delete

◆ Exemple avec Create

Create

Serie

Nom

AnneeDebut

AnneeFin

[Create](#)

[Back to List](#)

SerieID	Nom	AnneeDebut	AnneeFin
1	House Of The Dragon	2022	NULL
2	The Crown	2016	NULL
3	White Lotus	2021	NULL
4	The Witcher	2019	NULL
5	The Tudors	2007	2010



SerieID	Nom	AnneeDebut	AnneeFin
1	House Of The Dragon	2022	NULL
2	The Crown	2016	NULL
3	White Lotus	2021	NULL
4	The Witcher	2019	NULL
5	The Tudors	2007	2010
6	The boys	2019	NULL



❖ Insert (Create), Update et Delete

- ◆ Avec les contrôleurs et vues **auto-générés**, les **INSERT**, **UPDATE** et **DELETE** impactent toujours **une seule rangée** à la fois.
 - Si vous souhaitez pouvoir modifier **plusieurs rangées** d'un seul coup, il faudrait modifier une vue Razor et un contrôleur pour implémenter cette nouvelle opération.
 - Notez que si vous faites **plusieurs** **Add()**, **Update()** ou **Delete()** à un **DbSet** AVANT d'utiliser **.SaveChangesAsync()**, lorsque vous allez finalement appeler **.SaveChangesAsync()**, Entity Framework transformera tous vos changements en **un seul** **INSERT**, **UPDATE**, et / ou **DELETE** qui impactera donc **plusieurs rangées** dans la BD. (Plutôt que plusieurs **INSERT** individuels, par exemple)
 - Assurez-vous que vos **déclencheurs** dans la BD soient **capables de gérer des opérations sur plusieurs rangées** !
- ◆ À chaque fois que **.SaveChangesAsync()** est appelée, Entity Framework crée une **transaction** avec toutes les opérations effectuées depuis le dernier appel de **.SaveChangesAsync()**. Cette **transaction** réussie complètement ou échoue complètement. (Elle **rollback** entièrement s'il y a une erreur quelconque)



❖ Select (Retrieve)

- ◆ Nous allons revoir l'utilisation des opérations LINQ sur les DbSet.
 - Ce sera l'objet de notre prochaine rencontre



❖ Vues (Vues SQL, pas Vues Razor)

- ◆ Dans le **DbContext**, les vues SQL sont représentées par un **DbSet** comme n'importe quelle table !
 - On peut donc effectuer des requêtes **LINQ** sur ce **DbSet** sans problème.
 - Les **vues SQL** sont plus utiles que jamais car certaines requêtes LINQ peuvent se révéler complexes, mais si on crée une vue qui nous rassemble déjà les données (avec jointure ou agrégation par exemple), on a juste à se servir du DbSet de la vue SQL !

0 references

```
public virtual DbSet<VwDetailsSerie> VwDetailsSeries { get; set; }
```

- Le **controller** qui décrit les accès à ce **DbSet** (**Get** et **GetAll**) **DEVRAIT** omettre les opérations **Create, Update et Delete** car on ne veut pas faire de telles opérations sur les vues SQL.



❖ Vues (Vues SQL)

- ◆ Dans ce cas-ci, on va se servir de la vue **VW_DetailsSerie**, qui affiche les données des **séries**, accompagnées de trois données supplémentaires obtenues avec des agrégations et des jointures.

Cette requête est encapsulée par la vue :

```
WITH
Q1 AS (
    SELECT S.SerieID, COUNT(A.ActeurID) AS NbActeurs FROM Series.Serie S
    INNER JOIN Series.ActeurSerie A
    ON S.SerieID = A.SerieID
    GROUP BY S.SerieID
)
SELECT S.SerieID, S.Nom, S.AnneeDebut, S.AnneeFin,
Q1.NbActeurs,
SUM(SA.NbEpisodes) AS NbEpisodesTotal,
COUNT(SA.SaisonID) AS NbSaisons
FROM Series.Serie AS S
INNER JOIN Series.Saison AS SA
ON S.SerieID = SA.SerieID
INNER JOIN Q1
ON Q1.SerieID = S.SerieID
GROUP BY S.SerieID, S.Nom, S.AnneeDebut, S.AnneeFin, Q1.NbActeurs
```

```
SELECT * FROM dbo.VW_DetailsSerie;
```

SerieID	Nom	AnneeDebut	AnneeFin	NbActeurs	NbEpisodesTotal	NbSaisons
1	House Of The Dragon	2022	NULL	3	10	1
2	The Crown	2016	NULL	3	50	5
3	White Lotus	2021	NULL	3	13	2
4	The Witcher	2019	NULL	2	16	2
5	The Tudors	2007	2010	3	38	4



❖ Vues (Vues SQL)

- ◆ Bien entendu, comme la vue est toute prête, l'action du contrôleur est simple :

```
// GET: VwDetailsSeries
public async Task<IActionResult> IndexAvecViewSQL()
{
    return View(await _context.VwDetailsSeries.ToListAsync());
}
```

Copiez la vue Razor Index.cshtml pour l'appeler du même nom que votre nouvelle action.

La vue Razor reçoit une liste du Model qui représente la vue SQL :

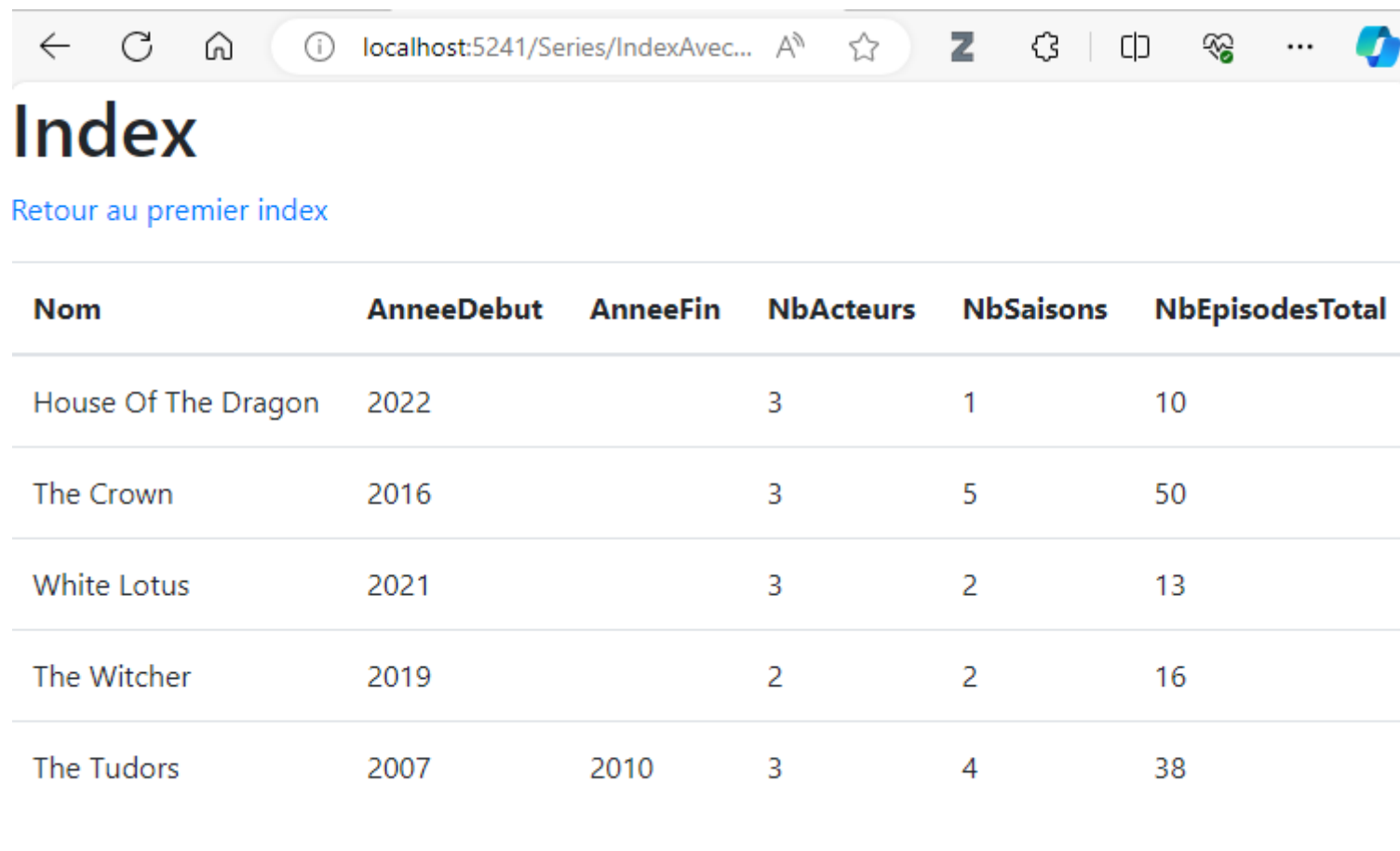
```
IndexAvecViewSQL.cshtml*  ↗ ✕
1  @model IEnumerable<Rencontre15.Models.VwDetailsSerie>
2
```

Ajustez les champs de la vue pour ajouter les 3 nouveaux champs.

Dans la vue, on enlève la possibilité de Créer, Supprimer ou Modifier un enregistrement car nos vues (SQL) ne servent qu'à voir les données.



❖ Vues (Vues SQL)



The screenshot shows a web browser window with the address bar displaying 'localhost:5241/Series/IndexAvec...'. The page title is 'Index'. Below the title is a link 'Retour au premier index'. The main content is a table with 6 columns: 'Nom', 'AnneeDebut', 'AnneeFin', 'NbActeurs', 'NbSaisons', and 'NbEpisodesTotal'. The table contains 5 rows of data.

Nom	AnneeDebut	AnneeFin	NbActeurs	NbSaisons	NbEpisodesTotal
House Of The Dragon	2022		3	1	10
The Crown	2016		3	5	50
White Lotus	2021		3	2	13
The Witcher	2019		2	2	16
The Tudors	2007	2010	3	4	38



❖ Procédures stockées

- ◆ Exemple : On a une procédure stockée qui nous donne toutes les séries d'un acteur dont on fournit le prénom et le nom :

```
CREATE PROCEDURE USP_ChcherSeriesParActeur
    @Prenom nvarchar(50),
    @Nom nvarchar(50)
AS
    SELECT V.SerieID, V.Nom, V.AnneeDebut, V.AnneeFin, V.NbActeurs, V.NbEpisodesTotal, V.NbSaisons
    FROM VW_DetailsSerie V
    INNER JOIN Series.ActeurSerie SA ON V.SerieID = SA.SerieID
    INNER JOIN Acteurs.Acteur AA ON SA.ActeurID = AA.ActeurID
    WHERE AA.Prenom = @Prenom AND AA.Nom = @Nom;
GO
```

```
EXEC dbo.USP_ChcherSeriesParActeur
@Prenom = 'Henry', @Nom = 'Cavill'
GO
```

SerieID	Nom	AnneeDebut	AnneeFin	NbActeurs	NbEpisodesTotal	NbSaisons
4	The Witcher	2019	NULL	2	16	2
5	The Tudors	2007	2010	3	38	4



❖ Procédures stockées

- ◆ Dans le contrôleur des séries (par exemple), on ajoute une méthode qui exécutera cette procédure stockée :

- Remarquez la manière de **passer les paramètres** à la **procédure stockée**. Cela permet d'éviter les **injections SQL**.

- Il faut être très prudent avec les risques d'injections SQL lorsqu'on fait exécuter du SQL pur à Entity Framework. Le type SqlParameter s'assure que la valeur fournie ne sera pas interprétée comme du code SQL exécutable. (Mais bien comme une simple valeur)

```
public async Task<IActionResult> SeriesParActeur(int id)
{
    Acteur? acteur = await _context.Acteurs.FindAsync(id);
    if (acteur == null)
    {
        return NotFound();
    }

    string query = "EXEC Series.UDP_ChercherSeriesParActeur @Prenom, @Nom";

    List<SqlParameter> parameters = new List<SqlParameter>
    {
        new SqlParameter{ParameterName="@Prenom", Value=acteur.Prenom},
        new SqlParameter{ParameterName="@Nom", Value=acteur.Nom}
    };

    List<VwDetailsSerie> series = await _context.VwDetailsSeries.FromSqlRaw(query, parameters.ToArray()).ToListAsync();

    return View(series);
}
```

- **DbContext.DbSet.FromSqlRaw()** permet d'aller récupérer un ensemble qui possède la **même structure que la table représentée par le DbSet utilisé**.

- Pour l'utilisation des SqlParameter, il se peut que vous deviez ajouter le paquet NuGet Microsoft.Data.SqlClient. La version la plus récente est correct. N'oubliez pas d'ajouter **using Microsoft.Data.SqlClient** en haut de votre contrôleur.



Microsoft.Data.SqlClient par Microsoft, 517M téléchargements 5.2.0
The current data provider for SQL Server and Azure SQL databases. This has replaced System.Data.SqlClient. These classes provide access to SQL and encapsulate database-specific protocols, including tabular data stream...



❖ Procédures stockées

- ◆ `DbContext.DbSet.FromSqlRaw()` permet d'aller récupérer un ensemble qui possède la **même structure que la table représentée par le DbSet utilisé**.

◆ ATTENTION:

- Cela signifie qu'avec `.FromSqlRaw()`, on ne peut pas appeler une procédure stockée qui retourne des données qui ne correspondent à aucune **table** / **vue** de la base de données.
- Il faut absolument que les données reçues puissent s'agencer à un **DbSet**.
- Si ce n'est pas le cas de votre procédure stockée, songez à ajouter une table juste pour avoir un DbSet (Nous verrons que nous ferons cela pour récupérer un champ dé-encrypté)



❖ Procédures stockées

- Cette action du contrôleur exécute la procédure stockée et retourne une liste de **VwDetailsSerie** à la vue.
- Nous aurions également pu créer un **ViewModel** qui contient un **Acteur** et une **List<VwDetailsSerie>** pour afficher les informations de l'acteur dans la vue, suivies par la liste de ses séries.

J'ai juste copié la vue faite précédemment (IndexAvecViewSQL.cshtml pour avoir le nom de la nouvelle action SeriesParActeur).
Pour faire simple, dans Index.cshtml, j'ai ajouté ce lien pour voir les séries de l'acteur 9

```
public async Task<IActionResult> SeriesParActeur(int id)
{
    Acteur? acteur = await _context.Acteurs.FindAsync(id);
    if (acteur == null)
    {
        return NotFound();
    }

    string query = "EXEC Series.USB_ChercherSeriesParActeur @Prenom, @Nom";

    List<SqlParameter> parameters = new List<SqlParameter>
    {
        new SqlParameter{ParameterName="@Prenom", Value=acteur.Prenom},
        new SqlParameter{ParameterName="@Nom", Value=acteur.Nom}
    };

    List<VwDetailsSerie> series = await _context.VwDetailsSerie.FromSqlRaw(query, parameters.ToArray()).ToListAsync();

    return View(series);
}
```



```
<p>
    <a asp-action="SeriesParActeur" asp-route-id="9">Voir les séries de l'acteur 9</a>
</p>
```



❖ Procédures stockées

```
public async Task<IActionResult> SeriesParActeur(int id)
{
    Acteur? acteur = await _context.Acteurs.FindAsync(id);
    if (acteur == null)
    {
        return NotFound();
    }

    string query = "EXEC Series.USB_ChercherSeriesParActeur @Prenom, @Nom";

    List<SqlParameter> parameters = new List<SqlParameter>
    {
        new SqlParameter{ParameterName="@Prenom", Value=acteur.Prenom},
        new SqlParameter{ParameterName="@Nom", Value=acteur.Nom}
    };

    List<VwDetailsSerie> series = await _context.VwDetailsSerie.FromSqlRaw(query, parameters.ToArray()).ToListAsync();
    return View(series);
}
```

Les séries de l'acteur dont l'id est 9:



Séries d'un acteur

Nom	AnneeDebut	AnneeFin	NbActeurs	NbSaisons	NbEpisodesTotal
The Witcher	2019		2	2	16
The Tudors	2007	2010	3	4	38



❖ Procédures stockées

◆ Procédure stockée qui ne retourne pas de données

- Par exemple, qui réalise des INSERT, des UPDATE ou des DELETE
- On utilise pour cela la méthode `ExecuteSqlRawAsync()` sur l'objet `Database`.
 - Cette méthode peut retourner le **nombre de rangées modifiées** dans la BD. (Si la procédure stockée ne possède pas l'instruction `SET NOCOUNT ON` !)

```
//EXEC PROC sans valeur de retour
0 références
public async Task ProcedureSansRetour()
{
    string query = "EXEC Acteurs.UDP_ProcedureSansRetour";
    await _context.Database.ExecuteSqlRawAsync(query);
    await _context.SaveChangesAsync();
}
```

- Des **paramètres SQL** peuvent être ajoutés après le string de la requête SQL comme dans l'exemple précédent.
- **Attention !** Si votre procédure **modifie (ou ajoute/supprime) des données** dans la BD, il faut ajouter `await _context.SaveChangesAsync()` après l'exécution.



❖ Procédures stockées

◆ Exécution de code SQL pur (« raw ») en général

- Ce type d'interaction avec la base de données peut présenter certains défis.
 - Plus grande prudence nécessaire face aux **injections SQL**.
 - Éventuelle **maintenance** à faire si une procédure (ou les tables qui lui sont associées) changent dans la base de données.
 - S'il y a une **erreur dans le string** qui représente l'instruction SQL, Entity Framework ne peut pas vraiment le détecter d'avance.

◆ Certaines librairies ou ORM permettent d'appeler les procédures stockées d'une manière plus élégante et encadrée.

- **ADO.NET** et **Dapper** en sont des exemples.



❖ Déclencheurs

- ◆ Au risque de ne pas vous surprendre... Bonne nouvelle : il n'y a rien à préparer / configurer pour pouvoir exploiter les triggers.
 - C'est bien entendu parce qu'ils sont définis par un mécanisme qui est **déjà automatique**.
 - Lorsqu'Entity Framework fait des INSERT / UPDATE / DELETE sur la base de données, les triggers s'activeront normalement.
 - **Petite nuance** : Rappelez-vous qu'Entity Framework n'exécute pas un INSERT / UPDATE / DELETE instantanément après qu'on a utilisé **.Add()**, **.Update()** ou **.Remove()** sur un DbSet.
 - C'est seulement après avoir fait **DbContext.SaveChangesAsync()** que le serveur SQL recevra les opérations INSERT / UPDATE / DELETE. (Si Entity Framework en a générées)