Rencontre 16

Application Web Database-First

Bases de données et programmation Web

Sommaire

- Interrogations de la BD:
  - Linq
  - Jointures
  - Vues

Interaction avec la BD

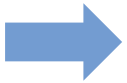
❖ Select (Retrieve)

♦ Quelques précisions pour les DbSet et les opérations retrieve

- Les DbSet, situés dans le DbContext et avec lesquels nos contrôleurs interagissent ne sont pas une « copie » des tables dans la BD. (Ça voudrait dire charger la BD en entier dans l'application Web 😬💀)
- Les DbSet sont simplement une « porte » ou une « **référence** » vers nos tables de la BD.
  - Cela dit, lorsqu'on fait ToListAsync() sur un DbSet, la table en **entier** est chargée dans l'appli ! Si on avait utilisé certains **filtres** avant de faire ToListAsync(), une quantité plus raisonnable de données aurait pu être chargée :

```
return await _context.Set<T>().ToListAsync();
```

• Ceci risque d'avoir un impact très mauvais sur la performance si la table récupérée est lourde.



```
return await _context.Set<T>().Take(50).ToListAsync();
```

• Ceci serait plus raisonnable. (Par exemple) Bien entendu, ne garder que les 50 premières rangées n'est pas forcément adapté à toutes les situations.

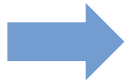
- C'est seulement au moment de faire ToListAsync() qu'Entity Framework envoie une requête SELECT à la BD pour convertir les données en liste C#. Donc plus on applique des filtres avant de faire ToListAsync(), mieux c'est pour limiter la quantité de données à charger.

Interaction avec la BD

❖ Select (Retrieve)

♦ Quelques précisions pour les DbSet et les opérations retrieve

```
return await _context.Set<T>().ToListAsync();
```



```
return await _context.Set<T>().Take(50).ToListAsync();
```

- Pas d'inquiétude ! Pour le moment utilisez la méthode de **gauche**. (Et appliquez les filtres **après**) Nous aborderons une meilleure stratégie lors de la semaine sur la performance.

Interaction avec la BD

❖ Select (Retrieve)

♦ Nous utiliserons des opérations LINQ sur les DbSet.

- Les opérations LINQ peuvent être réalisées dans les contrôleurs et dans les vues :

Dans un contrôleur

```
public async Task<IActionResult> FinishedSeries()
{
    IEnumerable<Serie> series = await _context.Series.ToListAsync();
    series = series.Where(x => x.AnneeFin != null);
    return View(series);
}
```

Dans une vue Razor

```
@foreach (var item in Model.Where(x => x.AnneeFin != null))
```

```
@model IEnumerable<Sem09.Models.Serie>
```

N'oubliez pas d'envoyer une **liste** de la donnée de votre choix à la vue via le **@model** pour pouvoir utiliser des opérations LINQ dans la vue !

- Rappel : Les opérations LINQ peuvent être **enchaînées** sur la même expression.

```
series = series.Where(x => x.AnneeDebut > 2010).OrderBy(x => x.Nom).Take(5);
```

Interaction avec la BD

❖ Select (Retrieve)

♦ Where

LINQ

```
series = series.Where(x => x.AnneeFin != null);
```

- La notation de **fonction anonyme** (**a** => instruction avec **a**) permet de vérifier une condition avec toutes les rangées du DbSet.
- Cette notation est utilisable pour une grande quantité d'instructions LINQ.

```
series = series.Where(x => x.AnneeFin != null && x.AnneeDebut < 2020);
```

SQL

```
SELECT * FROM Series.Serie
WHERE AnneeFin IS NOT NULL;
```

```
SELECT * FROM Series.Serie
WHERE AnneeFin IS NOT NULL AND AnneeDebut < 2020;
```

Interaction avec la BD

❖ Select (Retrieve)

♦ Order By

LINQ

```
series = series.OrderBy(x => x.AnneeDebut)
               .ThenByDescending(x => x.AnneeFin);
```

- OrderBy() est ascendant
- OrderByDescending() est descendant

- ThenBy() et ThenByDescending() sont optionnels, pour des tris secondaires.

SQL

```
SELECT * FROM Series.Serie
ORDER BY AnneeDebut ASC, AnneeFin DESC;
```

Interaction avec la BD

❖ Select (Retrieve)

♦ Skip et Take : Utilisés fréquemment pour la pagination (afficher x rangées à la fois)

LINQ

```
series = series.Take(5);
```

- Ne garde que les 5 premières rangées

```
series = series.Skip(5).Take(5);
```

- Ne garde que les rangées 6 à 10. (Les 5 premières ont été sautées)

SQL

```
SELECT TOP 5 * FROM Series.Serie;
```

- Un peu moins intuitif en SQL. Ci-dessous, 2 manières de le faire :

```
SELECT * FROM Series.Serie
ORDER BY Nom -- ORDER BY obligatoire pour offset/fetch
OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY;

SELECT TOP 10 * from Series.Serie
EXCEPT
SELECT TOP 5 * FROM Series.Serie
```

Interaction avec la BD

❖ Select (Retrieve)

♦ First, Last et Single : Pour récupérer un seul élément précis

- Les variantes avec *OrDefault* sont capables de retourner *null* si aucun élément n'est trouvé au lieu de faire **planter** le contrôleur.

```
Serie s = series.First();
```

```
Serie s = series.Last();
```

```
Serie s = series.Where(x => x.SerieId == 3).Single();
```

• Au lieu d'utiliser single() pour trouver avec l'id, peut-être que vous auriez pu utiliser le Get() (Retrieve one) du **service** ?

```
Serie? serie = await _serieService.Get(id);
if (serie == null)
{
    return NotFound();
}
```

```
public async Task<T?> Get(int id)
{
    return await _context.Set<T>().FindAsync(id);
}
```

```
Serie? s = series.FirstOrDefault();
```

```
Serie? s = series.LastOrDefault();
```

```
Serie? s = series.Where(x => x.SerieId == 3).SingleOrDefault();
```

• Cela dit, la Vue doit s'attendre à peut-être recevoir *null* au lieu des données !

(Ou bien le contrôleur pourrait vérifier si l'élément est *null* pour charger une vue différente)

```
@if (Model != null)
{
    <ul>
        <li>Id de la fleur : @Model.FlowerId</li>
        <li>Abeille responsable : @Model.GathererBee.Name</li>
        <li>Reine gestionnaire : @Model.GathererBee.QueenBee.Name</li>
        <li>Type : @Model.Type</li>
        <li>Âge (jours) : @Model.Age</li>
        <li>Pollen (mg) : @Model.Pollen</li>
        <li>Dernière pollinisation : @Model.LastPickup</li>
    </ul>
}
else
{
    <p>Nous n'avons pas trouvé de fleur ...</p>
}
```

Interaction avec la BD

❖ Select (Retrieve)

♦ Fonctions d'agrégation

LINQ

```
int count = series.Count();
```

```
double avg = series.Average(x => x.AnneeDebut);
```

```
double sum = series.Sum(x => x.AnneeDebut);
```

```
int max = series.Max(x => x.AnneeDebut);
```

```
int min = series.Min(x => x.AnneeDebut);
```

SQL

```
SELECT COUNT(*) FROM Series.Serie;
```

```
SELECT AVG(AnneeDebut) FROM Series.Serie;
```

Interaction avec la BD

❖ Select (Retrieve)

- ♦ Any() et All() : ce ne sont pas des commandes comparables à any et all en SQL.
- ♦ Dans Linq ces méthodes retournent un **booléan** qui nous dis si tous…ou un des….est là.

```
bool tousTerminées = series.All(x => x.AnneeFin != null);
```

- Est-ce que toutes les séries ont une année de fin ?

```
bool auMoinsUneTerminée = series.Any(x => x.AnneeFin != null);
```

- Est-ce qu'au moins une série possède une année de fin ?

Interaction avec la BD

❖ Select (Retrieve)

- ♦ Select() : Nous oblige à préparer un ViewModel adapté aux colonnes restantes si on souhaite l'envoyer dans une Vue. (Un objet anonyme introduirait de l'incertitude qui n'est pas bienvenue)

```
IEnumerable<SerieViewModel> seriesSimplifiees = series.Select(x => new SerieViewModel(x.Nom, x.AnneeDebut));
```

```
SELECT Nom, AnneeDebut FROM Series.Serie;
```

- L'équivalent en SQL. Beaucoup plus simple ici !

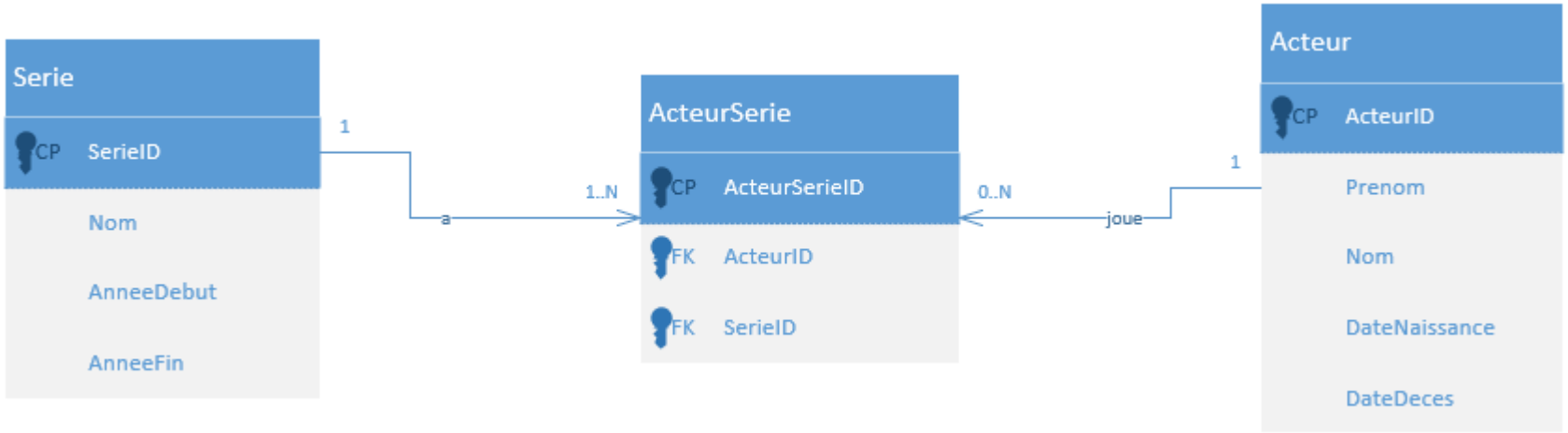
ViewModels
SerieViewModel.cs

```
public class SerieViewModel
{
    1 reference
    public string Nom { get; set; }
    1 reference
    public int AnneeDebut { get; set; }

    0 references
    public SerieViewModel(string nom, int anneeDebut) {
        Nom = nom;
        AnneeDebut = anneeDebut;
    }
}
```

Interaction avec la BD

❖ Jointures

- ♦ SQL et C# commencent à ne plus se comprendre ! Les jointures nous demanderont parfois un peu de préparation.
- ♦ Pour cet exemple, nous ferons une jointure entre Serie, ActeurSerie et Acteur pour avoir la **liste de tous les acteurs de chaque série**.



❖ Jointures

- ♦ Côté SQL, la requête serait « plutôt simple »... mais est-ce que le tableau obtenu est vraiment un affichage intéressant pour l'utilisateur ? *Bof*.
  - Les noms de séries qui se **répètent**...
  - Les séries qui s'**entremêlent** si on n'utilise pas ORDER BY...

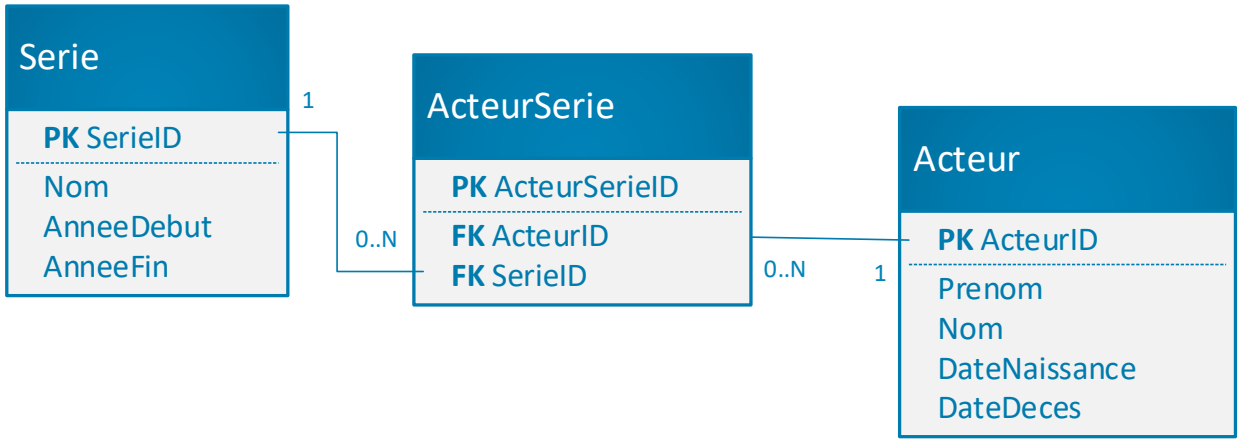
```
SELECT S.Nom, A.Prenom, A.Nom FROM Series.Serie S
INNER JOIN Series.ActeurSerie A_S
ON S.SerieID = A_S.SerieID
INNER JOIN Acteurs.Acteur A
ON A.ActeurID = A_S.ActeurID
```

Nom	Prenom	Nom
House Of The Dragon	Emma	D'Arcy
House Of The Dragon	Matt	Smith
The Crown	Matt	Smith
House Of The Dragon	Olivia	Cooke
The Crown	Claire	Foy
The Crown	Imelda	Staunton
White Lotus	Jennifer	Coolidge
White Lotus	Aubrey	Plaza
White Lotus	Alexandra	Daddario
The Witcher	Henry	Cavill
The Tudors	Henry	Cavill
The Witcher	Liam	Hemsworth
The Tudors	Natalie	Dormer
The Tudors	Jonathan	Rhys-Meyers

Interaction avec la BD

❖ Jointures

- ♦ On doit commencer par réfléchir à ce qu'on souhaite afficher et comment on veut l'afficher.
  - ○ On veut peut-être juste la liste des acteurs pour **une série en particulier** ?
  - ○ Si on veut vraiment toutes les séries et tous leurs acteurs, il faudra réfléchir à un **affichage plus élégant**.



Interaction avec la BD

❖ Jointures

♦ Exemple 1 : **La liste d'acteurs pour une seule série particulière**

- Qu'est-ce qu'on va envoyer à la vue ? Juste IEnumerable<Acteur> ou bien IEnumerable<Acteur> ET une Série ? (Celle dont on a demandé les acteurs)
  - Optons pour la **2e option** et préparons un **ViewModel** et une **Vue** en conséquence.

ViewModel

```
public class ActeursSerieViewModel
{
    8 references
    public Serie Serie { get; set; }
    5 references
    public IEnumerable<Acteur> Acteurs { get; set; }
    0 references
    public ActeursSerieViewModel(Serie serie, IEnumerable<Acteur> acteurs) {
        Serie = serie;
        Acteurs = acteurs;
    }
}
```

Vue qui recevra le ViewModel

ActeursSerie.cshtml | ActeursSerieViewModel.cs | SeriesControlle

```
@model Sem09.ViewModels.ActeursSerieViewModel;
@{
    ViewData["Title"] = "ActeursSerie";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Infos de la série dans la Vue

```
<p>
    @Html.DisplayNameFor(model => model.Serie.Nom) :
    @Html.DisplayFor(model => model.Serie.Nom)
</p>
<p>
    @Html.DisplayNameFor(model => model.Serie.AnneeDebut) :
    @Html.DisplayFor(model => model.Serie.AnneeDebut)
</p>
<p>
    @Html.DisplayNameFor(model => model.Serie.AnneeFin) :
    @(Model.Serie.AnneeFin == null ? "Série encore en ondes" :
    @Html.DisplayFor(model => model.Serie.AnneeFin))
</p>
```

Acteurs de la série dans la Vue

```
@foreach (var item in Model.Acteurs)
{
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Prenom)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Nom)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.DateNaissance)
        </td>
    </tr>
}
```

Interaction avec la BD

- On ne va pas vraiment faire une jointure ICI
  - Exemple 1 : La liste d'acteurs pour une seule série particulière
    - Nous allons utiliser UNE ALTERNATIVE pour simplifier,

UTILISABLE UNIQUEMENT S'IL Y A PEU DE DONNÉES

• Parfois, on peut prendre un peu de recul sur le mode opératoire SQL et simplifier beaucoup les choses à l'aide des outils fournis par LINQ !

• Ici, au lieu de faire une jointure, on a simplement utilisé .Where() et .Any(). Chaque acteur possède une List<ActeurSerie>. Chaque ActeurSerie est associé à un ActeurId et une SerieId. On a donc gardé que les acteurs qui possèdent une rangée dans ActeurSerie qui est associée à la bonne série.

• Vous êtes encouragés à utiliser ce genre de raccourcis lorsque c'est possible. Si on avait bel et bien eu besoin des informations contenues dans ActeurSerie, la jointure aurait été pertinente cela dit.

```
0 references
public async Task<IActionResult> ActeursSerie(int id)
{
    Serie? serie = await _context.Series.FindAsync(id);
    if (serie == null)
    {
        return NotFound();
    }

    IEnumerable<Acteur> acteurs = await _context.Acteurs.ToListAsync();
    IEnumerable<ActeurSerie> acteurSeries = await _context.ActeurSeries.ToListAsync();

    IEnumerable<Acteur> acteursDeLaSerie = acteurs
                .Where(a => a.ActeurSeries.Any(s => s.SerieId == serie.SerieId));

    return View(new ActeursSerieViewModel(serie, acteursDeLaSerie));
}
```

Remarquez que c'était aussi possible en SQL !

```
SELECT Prenom, Nom, DateNaissance, DateDeces FROM Acteurs.Acteur
WHERE ActeurID = ANY(SELECT ActeurID FROM Series.ActeurSerie WHERE SerieID = 1);
```

Interaction avec la BD

❖ Jointures

♦ Lazy Loading

- Par défaut, Entity Framework essaye d'être économe lorsqu'il charge des données de la base de données dans l'application Web.

```
public partial class Acteur
{
    [Key]
    [Column("ActeurID")]
    1 reference
    public int ActeurId { get; set; }

    [StringLength(50)]
    4 references
    public string Prenom { get; set; } = null!;

    [StringLength(50)]
    4 references
    public string Nom { get; set; } = null!;

    [Column(TypeName = "date")]
    4 references
    public DateTime DateNaissance { get; set; }

    [Column(TypeName = "date")]
    0 references
    public DateTime? DateDeces { get; set; }

    [InverseProperty("Acteur")]
    3 references
    public virtual ICollection<ActeurSerie> ActeurSeries { get; } = new List<ActeurSerie>();
```

- Si on n'utilisait pas le Lazy Loading, ça voudrait dire que **systématiquement**, demander la liste de tous les ActeurSerie (par exemple), forcerait **Entity Framework** à également charger tous les acteurs et toutes les séries de la BD ! C'est du **Eager Loading**.

- **C'est à tout prix à éviter**. Cela ralentirait systématiquement les accès à la base de données pour les tables Acteur, Serie et ActeurSerie.

```
public partial class ActeurSerie
{
    [Key]
    [Column("ActeurSerieID")]
    1 reference
    public int ActeurSerieId { get; set; }

    [Column("ActeurID")]
    0 references
    public int ActeurId { get; set; }

    [Column("SerieID")]
    2 references
    public int SerieId { get; set; }

    [ForeignKey("ActeurId")]
    [InverseProperty("ActeurSeries")]
    1 reference
    public virtual Acteur Acteur { get; set; } = null!;

    [ForeignKey("SerieId")]
    [InverseProperty("ActeurSeries")]
    1 reference
    public virtual Serie Serie { get; set; } = null!;
}
```

Interaction avec la BD

❖ Jointures

♦ Lazy Loading

- Une solution possible est de charger, **au cas par cas**, les tables dont on a besoin pour que notre **jointure** (ou toute autre opération multi-tables) fonctionne.

• Ceci ne fonctionne pas. a.ActeurSeries sera systématiquement ***null*** à cause du Lazy Loading et au final aucun acteur ne respectera la condition du Where().

```
IEnumerable<Acteur> acteurs = await _context.Acteurs.ToListAsync();
//IEnumerable<ActeurSerie> acteurSeries = await _context.ActeurSeries.ToListAsync();

IEnumerable<Acteur> acteursDeLaSerie = acteurs
                .Where(a => a.ActeurSeries.Any(s => s.SerieId == serie.SerieId));
```

• Ceci fonctionne. On a malheureusement dû demander les données de la table ActeurSerie, mais désormais, la référence ActeurSeries dans chaque Acteur est remplie et utilisable.

```
IEnumerable<Acteur> acteurs = await _context.Acteurs.ToListAsync();
IEnumerable<ActeurSerie> acteurSeries = await _context.ActeurSeries.ToListAsync();

IEnumerable<Acteur> acteursDeLaSerie = acteurs
                .Where(a => a.ActeurSeries.Any(s => s.SerieId == serie.SerieId));
```

Nous reparlerons d'optimisation de requête et de performance dans quelques semaines.

Interaction avec la BD

❖ Jointures

♦ Lazy Loading

- INSTALLEZ le package EntityFrameworkCore.Proxies


```
builder.Services.AddDbContext<SeriesTvContext>(
    options => {
        options.UseSqlServer(builder.Configuration.GetConnectionString("BDSeriesTV"));
        options.UseLazyLoadingProxies();
    });
```

- À partir du moment où on configure .UseLazyLoadingProxies() sur le DbContext, même si le Lazy Loading **reste actif**, quand on accède à une autre table via une propriété de référence avec le mot-clé virtual, Entity Framework vérifiera si cette table est chargée, et si non, la chargera pour s'assurer que tout fonctionne !

• Ceci deviendrait fonctionnel !

• La table ActeurSerie serait chargée dès le moment où on essaye d'y accéder avec a.ActeurSeries, par exemple.

```
IEnumerable<Acteur> acteurs = await _context.Acteurs.ToListAsync();
//IEnumerable<ActeurSerie> acteurSeries = await _context.ActeurSeries.ToListAsync();

IEnumerable<Acteur> acteursDeLaSerie = acteurs
                .Where(a => a.ActeurSeries.Any(s => s.SerieId == serie.SerieId));
```

Interaction avec la BD

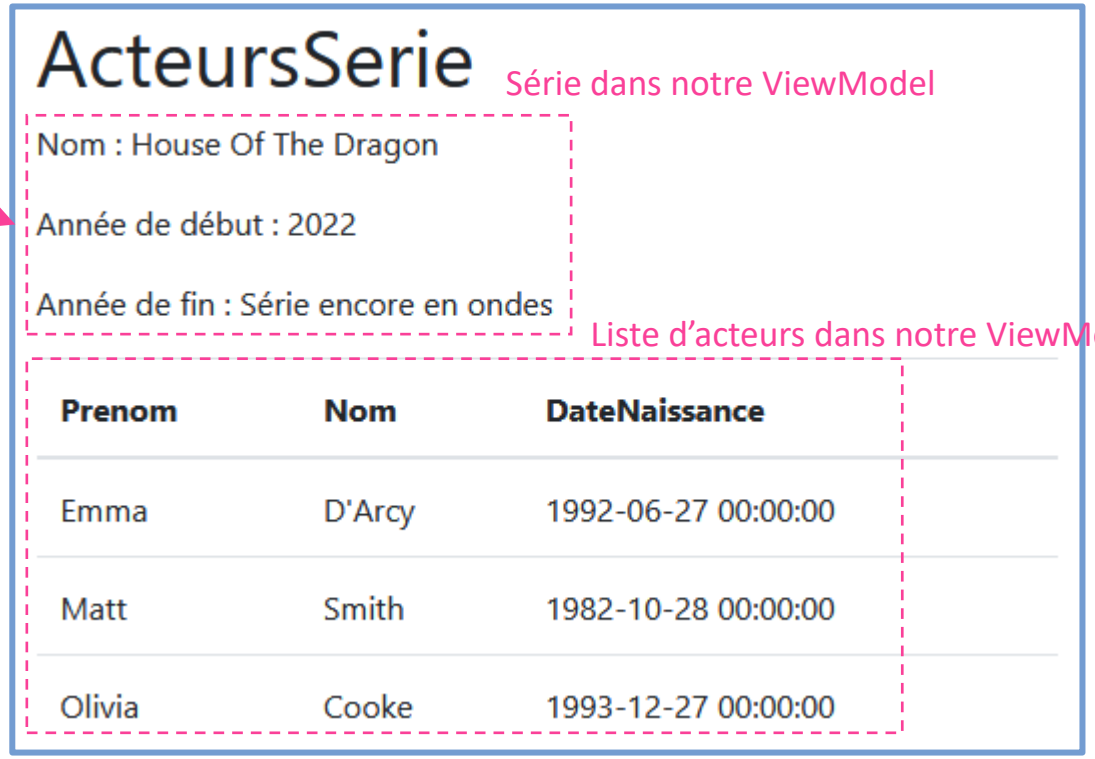
❖ Jointures

♦ Exemple 1 : La liste d'acteurs pour une seule série particulière

○ Résultat

```
<p>
    @Html.DisplayNameFor(model => model.Serie.Nom) :
    @Html.DisplayFor(model => model.Serie.Nom)
</p>
<p>
    @Html.DisplayNameFor(model => model.Serie.AnneeDebut) :
    @Html.DisplayFor(model => model.Serie.AnneeDebut)
</p>
<p>
    @Html.DisplayNameFor(model => model.Serie.AnneeFin) :
    @(Model.Serie.AnneeFin == null ? "Série encore en ondes" :
    @Html.DisplayFor(model => model.Serie.AnneeFin))
</p>
```

```
<thead>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Acteurs.First().Prenom)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Acteurs.First().Nom)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Acteurs.First().DateNaissance)
        </th>
    </tr>
</thead>
<tbody>
    @foreach (var item in Model.Acteurs)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Prenom)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Nom)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.DateNaissance)
            </td>
        </tr>
    }
</tbody>
```



Interaction avec la BD

❖ Jointures

♦ Exemple 2 : **Tous les acteurs de chaque série**

- Cette fois-ci, ce qu'on voudra envoyer à la vue est un IEnumerable<ActeursSerieViewModel>, donc la même chose que dans l'exemple précédent, mais plusieurs fois car on veut la liste pour chaque série.
- On pourrait donc faire à peu près la même chose, sauf qu'on ajoute une boucle pour faire un ActeursSerieViewModel par série.

Code pour l'exemple 1

```
0 références
public async Task<IActionResult> ActeursSerie(int id)
{
    Serie? serie = await _context.Series.FindAsync(id);
    if (serie == null)
    {
        return NotFound();
    }

    IEnumerable<Acteur> acteurs = await _context.Acteurs.ToListAsync();
    IEnumerable<ActeurSerie> acteurSeries = await _context.ActeurSeries.ToListAsync();

    IEnumerable<Acteur> acteursDeLaSerie = acteurs
                .Where(a => a.ActeurSeries.Any(s => s.SerieId == serie.SerieId));

    return View(new ActeursSerieViewModel(serie, acteursDeLaSerie));
}
```

Code pour l'exemple 2

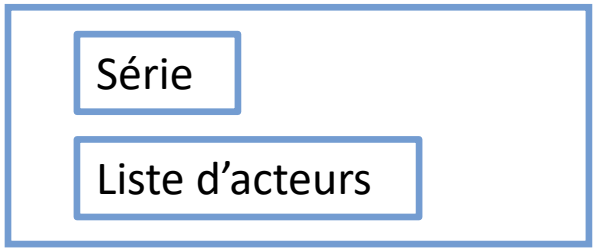
```
0 références
public async Task<IActionResult> ActeursSeries()
{
    IEnumerable<Serie> series = await _context.Series.ToListAsync();
    IEnumerable<Acteur> acteurs = await _context.Acteurs.ToListAsync();
    IEnumerable<ActeurSerie> acteurSeries = await _context.ActeurSeries.ToListAsync();

    List<ActeursSerieViewModel> acteursSeries = new List<ActeursSerieViewModel>();

    foreach (Serie serie in series)
    {
        IEnumerable<Acteur> acteursDeLaSerie = acteurs
                    .Where(a => a.ActeurSeries.Any(s => s.SerieId == serie.SerieId));
        acteursSeries.Add(new ActeursSerieViewModel(serie, acteursDeLaSerie));
    }

    return View(acteursSeries);
}
```

Interaction avec la BD

❖ Pour être clair : ce qu'on passe dans la vue

Exemple 1 : vue ActeursSerie

ActeursSerieViewModel



Exemple 2 : vue ActeursSeries

IEnumerable<ActeursSerieViewModel>

Interaction avec la BD

❖ Jointures

♦ Exemple 2 : Tous les acteurs de chaque série

- Pourquoi le DbSet ActeurSerie est récupéré « pour rien » dans les deux exemples ?

Code pour l'exemple 1

```
0 références
public async Task<IActionResult> ActeursSerie(int id)
{
    Serie? serie = await _context.Series.FindAsync(id);
    if (serie == null)
    {
        return NotFound();
    }

    IEnumerable<Acteur> acteurs = await _context.Acteurs.ToListAsync();
    IEnumerable<ActeurSerie> acteurSeries = await _context.ActeurSeries.ToListAsync();

    IEnumerable<Acteur> acteursDeLaSerie = acteurs
                .Where(a => a.ActeurSeries.Any(s => s.SerieId == serie.SerieId));

    return View(new ActeursSerieViewModel(serie, acteursDeLaSerie));
}
```

Code pour l'exemple 2

```
0 références
public async Task<IActionResult> ActeursSeries()
{
    IEnumerable<Serie> series = await _context.Series.ToListAsync();
    IEnumerable<Acteur> acteurs = await _context.Acteurs.ToListAsync();
    IEnumerable<ActeurSerie> acteurSeries = await _context.ActeurSeries.ToListAsync();

    List<ActeursSerieViewModel> acteursSeries = new List<ActeursSerieViewModel>();

    foreach (Serie serie in series)
    {
        IEnumerable<Acteur> acteursDeLaSerie = acteurs
                    .Where(a => a.ActeurSeries.Any(s => s.SerieId == serie.SerieId));
        acteursSeries.Add(new ActeursSerieViewModel(serie, acteursDeLaSerie));
    }

    return View(acteursSeries);
}
```

- Cette liste n'est jamais accédée / utilisée directement dans les deux exemples.
- On pourrait la retirer sans problème et le code compilerait et s'exécuterait sans bogue.

- Pourquoi doit-on absolument la garder ? -> Lazy Loading !

Interaction avec la BD

❖ Jointures

♦ Lazy Loading

- Par défaut, Entity Framework essaye d'être économe lorsqu'il charge des données de la base de données dans l'application Web.

```
public partial class Acteur
{
    [Key]
    [Column("ActeurID")]
    1 reference
    public int ActeurId { get; set; }

    [StringLength(50)]
    4 references
    public string Prenom { get; set; } = null!;

    [StringLength(50)]
    4 references
    public string Nom { get; set; } = null!;

    [Column(TypeName = "date")]
    4 references
    public DateTime DateNaissance { get; set; }

    [Column(TypeName = "date")]
    0 references
    public DateTime? DateDeces { get; set; }

    [InverseProperty("Acteur")]
    3 references
    public virtual ICollection<ActeurSerie> ActeurSeries { get; } = new List<ActeurSerie>();
```

- Par exemple, disons qu'on veut récupérer la liste de tous les acteurs de la BD. Il faut garder à l'esprit que puisque la référence vers la table ActeurSeries dans notre Model Acteur est marquée comme « **virtual** », cette donnée sera « **lazy loadée** » : Cette donnée sera ***null*** si on demande juste la liste des acteurs du DbSet d'acteurs.

- C'est la même chose pour l'entité ActeurSerie : si on récupère la liste des ActeurSerie dans la BD, les références Acteur et Serie seront toutes ***null***.

```
public partial class ActeurSerie
{
    [Key]
    [Column("ActeurSerieID")]
    1 reference
    public int ActeurSerieId { get; set; }

    [Column("ActeurID")]
    0 references
    public int ActeurId { get; set; }

    [Column("SerieID")]
    2 references
    public int SerieId { get; set; }

    [ForeignKey("ActeurId")]
    [InverseProperty("ActeurSeries")]
    1 reference
    public virtual Acteur Acteur { get; set; } = null!;

    [ForeignKey("SerieId")]
    [InverseProperty("ActeurSeries")]
    1 reference
    public virtual Serie Serie { get; set; } = null!;
}
```

Interaction avec la BD

❖ Select (Retrieve)

- ♦ GroupBy() : Comme pour SELECT ou une Jointure, on s'apprête à envoyer à la vue une donnée qui ne correspond pas à un de nos Models, alors on doit préparer un ViewModel.
  - Exemple : Afficher la liste des séries AINSI QUE le nombre d'acteurs pour chaque série. (Le nombre d'acteurs sera obtenu à l'aide de COUNT et GROUP BY)

Index avec nombre d'acteurs

Nom	Année de début	Année de fin	Nombre d'acteurs
House Of The Dragon	2022		3
The Crown	2016		3
White Lotus	2021		3
The Witcher	2019		2
The Tudors	2007	2010	3

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

```sql
SELECT S.Nom, S.AnneeDebut, S.AnneeFin, COUNT(A_S.ActeurID) AS 'NbActeurs'
FROM Series.Serie S
INNER JOIN Series.ActeurSerie A_S
ON A_S.SerieID = S.SerieID
GROUP BY S.Nom, S.AnneeDebut, S.AnneeFin
```

ViewModel

```
public class SerieAvecNbActeursViewModel
{
    7 references
    public Serie Serie { get; set; }

    [DisplayName("Nombre d'acteurs")]
    3 references
    public int NbActeurs { get; set; }

    0 references
    public SerieAvecNbActeursViewModel(Serie serie, int nbActeurs) {
        Serie = serie;
        NbActeurs = nbActeurs;
    }
}
```

Type attendu par la vue

```
@model IEnumerable<Sem09.ViewModels.SerieAvecNbActeursViewModel>
```

Interaction avec la BD

❖ Select (Retrieve)

- ♦ GroupBy() ... sans GroupBy()
  - ○ Il est plutôt facile de substituer notre requête pour quelque chose d'un peu plus léger dans ce cas spécifique :

Constructeur du ViewModel

```
1 reference
public SerieAvecNbActeursViewModel(Serie serie, int nbActeurs) {
    Serie = serie;
    NbActeurs = nbActeurs;
}
```

```
0 references
public async Task<IActionResult> NbActeursParSerie()
{
    IEnumerable<Serie> series = await _context.Series.ToListAsync();
    IEnumerable<ActeurSerie> acteurSeries = await _context.ActeurSeries.ToListAsync();

    IEnumerable<SerieAvecNbActeursViewModel> seriesAvecNbActeurs = series
        .Select(x => new SerieAvecNbActeursViewModel(x, x.ActeurSeries.Count));

    return View(seriesAvecNbActeurs);
}
```

- Avec Select(), on a pu simplement restructurer le résultat en le convertissant en notre ViewModel. Le **constructeur du ViewModel** fonctionne car x est bel et bien une **Serie** et x.ActeurSeries.Count est bel et bien un **int** !

- Comme d'habitude, attention au Lazy Loading !

- Ce n'est pas forcément pertinent ici, mais si on avait voulu remplacer le Count par la Sum() des ActeurId, on aurait fait comme ceci :

```
IEnumerable<SerieAvecNbActeursViewModel> seriesAvecNbActeurs = series
    .Select(x => new SerieAvecNbActeursViewModel(x, x.ActeurSeries.Sum(a => a.ActeurId)));
```

Interaction avec la BD

- 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 pouvent 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 ils ne nous intéressent pas avec les vues SQL.

Interaction avec la BD

❖ 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

Interaction avec la BD

❖ Vues (Vues SQL)

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

```
public async Task<IActionResult> IndexAvecVue()
{
    IEnumerable<VwDetailsSerie> series = await _context.VwDetailsSeries.ToListAsync();
    return View(series);
}
```

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

```
@model IEnumerable<Sem09.Models.VwDetailsSerie>
```

Index avec View

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

Les jointures ou une procédure/une vue

- Les jointures sont faites avec Linq ou la méthode syntaxe quand les données sont nombreuses.
- OU plus souvent encore, on a des procédures qui font les jointures et l'extraction au niveau de la BD, et on montre le résultat finalement.
- Ou une vue.
- Bref, pensez à utiliser la BD pour vous simplifier la vie!