


 Icône Ajouter Ajouter une couverture

Tutos MVC 2

Table des matières


Tuto 22/04

Tuto : Entity Framework - Code First

1. Qu'est-ce que le Code First ?
 2. Étapes principales
 3. Fichiers générés par EF
 4. Modifier le modèle par la suite
 5. Seed de données (valeurs initiales)
 6. Relations entre entités
 7. Supprimer une migration
 8. Bonnes pratiques
-  Exercice pratique : Créer un modèle complet avec relation + Seed + Migration
- Étapes :

Tuto 24/04













Generator c'est lui le plus fort !

- Ce que ça génère si O aux questions
-  Exemple de flux pour un modèle Societe.cs
- À partir de :
- Il génère :
- Pourquoi c'est utile ?
- À retenir


TUTO CosmosTrial

1. Créer la solution et les projets
2. Ajouter une référence entre les projets
3. Ajouter les packages NuGet nécessaires
4. Créer ton premier modèle
5. Créer un DbContext
6. Configurer le DbContext dans CosmosWeb
7. Générer la base de données (migrations)







Tuto 13/05 CosmosTrial

-  Tuto : Ajouter une entité liée et faire une migration
-  Étape 1 – Créer la classe Categorie
-  Étape 2 – Modifier la classe Produit pour ajouter la relation
-  Étape 3 – Ajouter une migration
-  Étape 4 – Appliquer la migration
-  Résultat
-  Comprendre la relation implicite avec Categorield
-  Exemple :
-  Que comprend EF avec ça ?
-  Pourquoi ça fonctionne ?
-  Tu peux aller plus loin (optionnel)
-  Bonus : tu peux le forcer avec Fluent API si nécessaire





Tuto 20/05 Seed

-  Initialiser une base Code First (Seed + Outils)
1. Seed de données de base
 2. Lien avec les migrations EF Core
 3. Gestion dans la migration
 4. Réinitialiser la base (en dev)
 5. Afficher les données seedées

Tuto 27/05 Formulaire

-  Objectif :
-  Étape 1 – Ajouter les paramètres à l'action Index
-  Étape 2 – Vue Index.cshtml
-  Étape 3 – Ajouter des liens de tri dans le tableau
-  Astuce : conserver les filtres dans les liens de tri
-  Résultat attendu :

Tuto 03/06 ModelState

- Objectifs :
-  1. Afficher un message si aucun élément
-  2. Utiliser TempData ou ViewBag pour afficher un message après action
- Dans le contrôleur (DeleteConfirmed, Create, etc.) :
- Dans la vue Index.cshtml (tout en haut) :
-  3. Gérer proprement NotFound et BadRequest
-  4. Affichage conditionnel plus fin
- Résumé

✓ 5. Validation des données avec ModelState.IsValid

Exemple complet dans Create :

En lien avec la vue :

Tuto 10/06 Suppression/Archivage d'un produit

🔗 Objectif :

Étape 1 – Ajouter le champ dans Produit.cs

Étape 2 – Nouvelle migration

Étape 3 – Modifier Index dans ProduitsController.cs

Étape 4 – Lien "Archiver" dans la vue Index.cshtml

Étape 5 – Nouvelle action dans ProduitsController.cs

Étape 6 – Afficher les produits archivés

✓ Résultat attendu

Tuto 17/06 Extensions

Pour reprendre ce qui a été dit au dernier tuto

Qu'est-ce qu'une méthode d'extension ?

Voici les fichiers Archivable

Voici les utilisations dans le contrôleur

D'autres idées d'extensions

Tuto 24/06 Migration

Cas 1 : Mauvais nom ou migration inutile

Solution

Cas 2 : Migration appliquée mais contenant une erreur

Scénario

Étapes :

Cas 3 : Repartir de zéro en local avec une base vide (reset de dev)

Scénario

Solution

🔗 Pourquoi c'est important

Bonus – Fichiers clés

QCM

Tuto 01/07 Migration en PROD

Cas classique chez un client

Étapes recommandées

1. Générer le SQL de migration côté dev

2. Vérifier / nettoyer le script SQL généré

3. Transmettre et appliquer côté client

Bonus : éviter les pièges

Cas 1 – Migration EF déjà appliquée partiellement par accident

Scénario

Solution recommandée

Cas 2 – Base de données Azure SQL

Particularité :

Solution :

Cas 3 – Multi-migrations à appliquer (le client a du retard)

Scénario

Solution :

Cas 4 – Rollback de migration EF Core

Scénario

Solution correcte (rollback réel)

Cas 5 – Gestion SQL Server et PostgreSQL côté dev

Tuto 08/07 Les champs persos et Entity Personnel

Tuto 22/04

🔧 Tuto : Entity Framework - Code First

1. Qu'est-ce que le Code First ?

Entity Framework en **Code First** signifie que tu **définis tes entités C# (modèles)** en premier, puis EF se charge de créer automatiquement la base de données à partir de ces classes.

C'est l'inverse du Database First, où l'on part d'une base existante.

2. Etapes principales

a. Créer une entité (le modèle)

```

1  using System.ComponentModel.DataAnnotations;
2
3  public class Produit
4  {
5      [Key]
6      public int Id { get; set; }
7
8      [Required(ErrorMessage = "Le nom est obligatoire.")]
9      [StringLength(100)]
10     public string Nom { get; set; }
11
12     [Range(0.01, 10000, ErrorMessage = "Le prix doit être supérieur à 0.")]
13     public decimal Prix { get; set; }
14 }

```

Voici un exemple un peu plus poussé avec des modèles parent

```

public class Personnel : EntityId
{
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string TelephonePersonnel { get; set; }
    public string TelephoneMobileProfessionnel { get; set; }
    public StatutContrat StatutContrat { get; set; }
    public DateTime DebutContrat { get; set; }
    public DateTime? FinContrat { get; set; }
}

public class StatutContrat : EntityId
{
    public int StatutContratId { get; set; }
    public string StatutContratNom { get; set; }
}

//En partant du principe que toutes les classes de l'appli ont une ID, même méthode si l'ID est un uuid
public class EntityId
{
    public int Id { get; set; }
}

//Sur une appli plus complexe, on pourrait aussi avoir une class EntityWithIdAndName voir d'autres du même genre pour uniformiser
public class EntityWithIdAndName
{
    public int Id { get; set; }
    public string Name { get; set; }
}

//Et donc avoir un Model StatutContrat comme ceci
public class StatutContrat : EntityWithIdAndName
{
}

```

b. Créer un DbContext

```

1  public class MonAppDbContext : DbContext
2  {
3      public DbSet<Produit> Produits { get; set; }
4
5      public MonAppDbContext(DbContextOptions<MonAppDbContext> options) : base(options) { }
6  }
7

```

c. Configurer EF dans Program.cs

</> C#

```
1 builder.Services.AddDbContext<MonAppDbContext>(options =>
2     options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
3
```

d. Créer une migration (ATTENTION : il y a toujours 2 façons pour les lignes de commande)

.NET CLI :

</> Shell

```
1 dotnet ef migrations add InitialCreate
2
```

Package Manager Console (PMC) :

PowerShell

```
1 Add-Migration InitialCreate
2
```

e. Appliquer la migration

.NET CLI :

Shell

```
1 dotnet ef database update
2
```

PMC :

PowerShell

```
1 Update-Database
2
```

3. Fichiers générés par EF

- `Migrations/20240425_InitialCreate.cs` : contient les instructions de création
- `ModelSnapshot.cs` : représente l'état actuel du modèle (comparaison future)

4. Modifier le modèle par la suite

Exemple : ajouter un champ `Stock`

C#

```
1 public int Stock { get; set; }
2
```

Puis : .NET CLI :

Shell

```

1 dotnet ef migrations add AddStockToProduit
2
3 dotnet ef database update
4

```

PMC :

```

1 Add-Migration AddStockToProduit
2
3 Update-Database
4

```

PowerShell

5. Seed de données (valeurs initiales)

Tu peux ajouter des données par défaut dans `OnModelCreating` :

```

1 protected override void OnModelCreating(ModelBuilder modelBuilder)
2 {
3     modelBuilder.Entity<Produit>().HasData(
4         new Produit { Id = 1, Nom = "Stylo", Prix = 1.50m },
5         new Produit { Id = 2, Nom = "Cahier", Prix = 2.00m }
6     );
7 }
8

```

C#

N'oublie pas de faire une migration pour que ces données soient gérées Tabernacle !

6. Relations entre entités

a. Exemple : Produit appartenant à une Catégorie

```

1 public class Produit
2 {
3     public int Id { get; set; }
4     public string Nom { get; set; }
5
6     public int CategorieId { get; set; }
7     public Categorie Categorie { get; set; }
8 }
9
10 public class Categorie
11 {
12     public int Id { get; set; }
13     public string Nom { get; set; }
14
15     public List<Produit> Produits { get; set; }
16 }
17

```

C#

EF déduit automatiquement la relation `one-to-many`.

b. Fluent API (facultatif pour personnaliser), c'est ce que fait (entre autre) EF Power Tools pour le Database First

```

1 modelBuilder.Entity<Produit>()
2     .HasOne(p => p.Categorie)
3     .WithMany(c => c.Produits)

```

C#

```
4     .HasForeignKey(p => p.CategorieId);  
5
```

7. Supprimer une migration

Avant d'avoir exécuté `update` : .NET CLI :

```
1 dotnet ef migrations remove  
2
```

Shell

PMC :

```
1 Remove-Migration  
2
```

PowerShell

Si la migration a déjà été appliquée, il faut d'abord exécuter : .NET CLI :

```
1 dotnet ef database update NomDeMigrationPrecedente  
2
```

Shell

PMC :

```
1 Update-Database NomDeMigrationPrecedente  
2
```

PowerShell

Puis :

```
1 dotnet ef migrations remove  
2
```

Shell

ou

```
1 Remove-Migration  
2
```

PowerShell

Ne jamais supprimer des migrations appliquées sans rollback de la base !

8. Bonnes pratiques

- Nommer les migrations clairement (`AddX`, `RemoveY`, `UpdateZ`)

- Toujours commit le dossier `Migrations/`
- Séparer les contextes si plusieurs bases ou modules
- Préférer `ToListAsync()` et `FirstOrDefaultAsync()` pour les appels EF
- Utiliser `AsNoTracking()` quand la modification des entités n'est pas nécessaire (perf)

C'est ICI qu'il faut réclamer le QCM !

Exercice pratique : Créer un modèle complet avec relation + Seed + Migration

Objectif :

Créer une entité `Client` liée à plusieurs `Commandes`, avec données de seed, migration et mise à jour de la base.

Étapes :

1. Créer les classes :

```
1 public class Client
2 {
3     public int Id { get; set; }
4     public string Nom { get; set; }
5
6     public List<Commande> Commandes { get; set; }
7 }
8
9 public class Commande
10 {
11     public int Id { get; set; }
12     public string Produit { get; set; }
13     public int Quantite { get; set; }
14
15     public int ClientId { get; set; }
16     public Client Client { get; set; }
17 }
18
```

1. Ajouter au `DbContext` :

```
1 public DbSet<Client> Clients { get; set; }
2 public DbSet<Commande> Commandes { get; set; }
3
```

1. Ajouter des données dans `OnModelCreating` :

```
1 modelBuilder.Entity<Client>().HasData(
2     new Client { Id = 1, Nom = "Alice" }
3 );
4
5 modelBuilder.Entity<Commande>().HasData(
6     new Commande { Id = 1, Produit = "Ordinateur", Quantite = 1, ClientId = 1 },
7     new Commande { Id = 2, Produit = "Souris", Quantite = 2, ClientId = 1 }
8 );
9
```


1. Créer une migration :

```
1 dotnet ef migrations add CreateClientCommande
2
```

Shell

ou

```
1 Add-Migration CreateClientCommande
2
```

PowerShell

1. Mettre à jour la base :

```
1 dotnet ef database update
2
```

Shell

ou

```
1 Update-Database
2
```

PowerShell

En bonus : créer une vue `Index` simple listant les clients et leurs commandes avec Razor.

Tuto 24/04

⚙️ Generator c'est lui le plus fort !

Il génère **tous les fichiers nécessaires** au bon fonctionnement de l'application, selon les conventions du projet **CosmosMVC** :

- Interface de service (ISocieteService)
- Implémentation du service (SocieteService)
- ViewModel (SocieteViewModel)
- Contrôleur (SocieteController)
- Vues Razor (Index, Edit, Delete)
- Créer MenuViewComponentGenerate.cs que le MenuViewComponent utilise pour générer un sous-menu de test
- Lignes à ajouter dans ServiceRegistration.cs

C'est un projet console C# dans la solution CosmosMVC.

1. Il **parcourt** tous les fichiers du dossier `CosmosModels\Models`
2. Il **ignore** certains modèles via une liste d'exceptions (`modelsException`)

3. Il extrait automatiquement les propriétés, attributs ([Key] , [Required] , etc.)
4. Il applique les conventions maison :
 - ViewModel avec ToModel() et ToEntity()
 - Controller héritant de BaseController
 - Vue _Edit.cshtml avec génération du formulaire

Ce que ça génère si O aux questions

```
Générer les services ? (O/N)

Pas de génération de services

Générer les contrôleurs ? (O/N)

Pas de génération de contrôleurs

Générer les viewmodels ? (O/N)

Pas de génération de viewmodels

Générer les views pour chaque modèle ? (O/N)

Pas de génération des views

Générer les lignes de menu pour 'MenuViewComponent' ? (O/N)

Pas de génération des entrées de menu
```

🔧 Exemple de flux pour un modèle Societe.cs

À partir de :

```
1 public class Societe
2 {
3     [Key]
4     public string Id { get; set; }
5
6     [Required]
7     public string Nom { get; set; }
8 }
9
```

C#

Il génère :

- ISocieteService.cs → interface de service
- SocieteService.cs → service dérivé de GenericService<Societe>
- SocieteViewModel.cs → contient les propriétés et les méthodes ToModel() et ToEntity()
- SocieteController.cs → hérite de BaseController<Societe, ISocieteService, SocieteViewModel>
- Les vues Razor : Index, Delete, _Edit
- Une ligne dans MenuViewComponent :

```
1 new() { Name = "Societe", Url = "/Societe", Icon = "file" }
2
```

C#

- Et une ligne à copier dans `ServiceRegistration.cs` :

```
1 services.AddScoped<ISocieteService, SocieteService>();  
2
```

C#

Pourquoi c'est utile ?

- Gain de temps énorme pour les nouveaux modèles
- Uniformisation totale du code généré
- Évite les oublis (vue manquante, `ToEntity()` oublié, etc.)
- Le code généré est prêt à modifier, pas à garder en l'état si besoin de logique personnalisée

À retenir

- Le code généré est **basique et modifiable** (ex : un champ spécial ou une vue custom peut être adaptée ensuite).
- Si tu **rajoutes une propriété à un modèle**, il faut régénérer le ViewModel ou la compléter à la main.
- **Toujours vérifier les ViewModels générés avant de les utiliser en prod.**

À ce moment du tuto on peut voir directement le code pour donner une idée de comment c'est fait (non pas de QCM aujourd'hui arrêtez de réclamer tout le temps comme ça, moi à Noël j'avais une orange et j'étais heureux !)

TUTO CosmosTrial

1. Créer la solution et les projets

1. Ouvrir Visual Studio → *Créer un nouveau projet*
2. Sélectionner **Bibliothèque de classes (.NET)**
 - Nom du projet : `CosmosModels`
 - Cocher *Créer une nouvelle solution* → nom de la solution : `CosmosLight`
3. Cliquer sur *Créer*
4. Ajouter un **nouveau projet** à la solution :
 - Type : **Application Web ASP.NET Core (MVC)**
 - Nom du projet : `CosmosWeb`
 - Dans le template, choisir **MVC**, pas d'authentification, .NET 8 ou celle utilisée

2. Ajouter une référence entre les projets

1. Clic droit sur le projet `CosmosWeb` → *Ajouter* → *Référence...*
2. Coche `CosmosModels`
3. Valider

3. Ajouter les packages NuGet nécessaires

Dans le projet **CosmosModels** :

1. Clic droit sur **CosmosModels** → *Gérer les packages NuGet*
2. Onglet **Parcourir**, chercher et installer :
 - **Microsoft.EntityFrameworkCore**
 - **Microsoft.EntityFrameworkCore.SqlServer**
 - **Microsoft.EntityFrameworkCore.Design**
 - **Microsoft.EntityFrameworkCore.Tools**

⚠ **Design** est requis pour générer les migrations

4. Créer ton premier modèle

Dans **CosmosModels**, ajouter un dossier **Models**, puis un fichier **Produit.cs** :

```
1 using System.ComponentModel.DataAnnotations;
2
3 namespace CosmosModels.Models
4 {
5     public class Produit
6     {
7         [Key]
8         public int Id { get; set; }
9
10        [Required]
11        public string Nom { get; set; }
12
13        [Range(0.01, 10000)]
14        public decimal Prix { get; set; }
15    }
16 }
17
```

C#

5. Créer un DbContext

Dans **CosmosModels**, ajouter un dossier **Context**, puis un fichier **CosmosLightContext.cs** :

```
1 using Microsoft.EntityFrameworkCore;
2 using CosmosModels.Models;
3
4 namespace CosmosModels.Context
5 {
6     public class CosmosLightContext : DbContext
7     {
8         public CosmosLightContext(DbContextOptions<CosmosLightContext> options) : base(options)
9         { }
10
11        public DbSet<Produit> Produits { get; set; }
12    }
13 }
```

C#

6. Configurer le DbContext dans CosmosWeb

1. Ouvrir `Program.cs` dans `CosmosWeb`
2. Ajouter en haut :

```
1 using CosmosModels.Context;  
2 using Microsoft.EntityFrameworkCore;  
3
```

C#

1. Dans `builder.Services`, ajouter :

```
1 builder.Services.AddDbContext<CosmosLightContext>(options =>  
2     options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));  
3
```

C#

1. Dans `appsettings.json`, ajouter une chaîne de connexion :

```
1 "ConnectionStrings": {  
2   "DefaultConnection": "Server=(localdb)\  
   \mssqllocaldb;Database=CosmosLightDB;Trusted_Connection=True;"  
3 }  
4
```

JSON

7. Générer la base de données (migrations)

1. Clic droit sur la solution → *Définir CosmosWeb comme projet de démarrage*
2. Ouvrir la console du gestionnaire de package (PMC) (Menu Affichage/Autres fenêtres)
3. Exécuter les commandes suivantes :

```
1 Add-Migration InitialCreate -Project CosmosModels  
2 Update-Database -Project CosmosModels  
3
```

PowerShell

Le `-Project` indique que les migrations sont dans `CosmosModels`, mais qu'on les exécute via le projet démarré (`CosmosWeb`).

Tuto 13/05 CosmosTrial

 Tuto : Ajouter une entité liée et faire une migration

📁 Étape 1 – Créer la classe Catégorie

Dans `CosmosModels/Models`, ajouter un fichier `Categorie.cs` :

```
1 using System.ComponentModel.DataAnnotations;
2
3 namespace CosmosModels.Models
4 {
5     public class Categorie
6     {
7         [Key]
8         public int Id { get; set; }
9
10        [Required]
11        public string Nom { get; set; }
12    }
13 }
14
```

C#

🔗 Étape 2 – Modifier la classe Produit pour ajouter la relation

Dans `Produit.cs`, ajoute les propriétés suivantes :

```
1 public int CategorieId { get; set; }
2 public Categorie Categorie { get; set; }
3
```

C#

Le fichier complet devient :

```
1 using System.ComponentModel.DataAnnotations;
2
3 namespace CosmosModels.Models
4 {
5     public class Produit
6     {
7         [Key]
8         public int Id { get; set; }
9
10        [Required]
11        public string Nom { get; set; }
12
13        [Range(0.01, 10000)]
14        public decimal Prix { get; set; }
15
16        public int CategorieId { get; set; }
17        public Categorie Categorie { get; set; }
18    }
19 }
20
```

C#

🔄 Étape 3 – Ajouter une migration

1. Ouvrir la console du gestionnaire de package (PMC)
2. Lancer la commande suivante :

```
1 Add-Migration AddCategorieAndRelation -Project CosmosModels
2
```

PowerShell

🌱 Étape 4 – Appliquer la migration

Toujours dans la console PMC :

PowerShell

```
1 Update-Database -Project CosmosModels
2
```

📌 Résultat

Tu obtiens :

- Une table `Categorie` avec `Id` et `Nom`
- Une table `Produit` avec une colonne `CategorieId` et une **relation (foreign key)** entre les deux

EF Core a automatiquement compris la relation grâce aux conventions de nommage.

🧠 Comprendre la relation implicite avec CategorieId

Dans Entity Framework Core, tu n'as **pas besoin d'écrire manuellement une contrainte de clé étrangère** ou d'utiliser Fluent API pour chaque relation simple. EF utilise les **conventions de nommage** pour déduire automatiquement les relations entre entités.

🔄 Exemple :

C#

```
1 public class Produit
2 {
3     public int CategorieId { get; set; } // Clé étrangère
4     public Categorie Categorie { get; set; } // Navigation
5 }
6
```

✅ Que comprend EF avec ça ?

1. `CategorieId` est une **clé étrangère** pointant vers une entité `Categorie`
2. `Categorie` est une **propriété de navigation**
3. EF en déduit une **relation one-to-many** :
 - Un **Produit** appartient à une **seule Catégorie**
 - Une **Catégorie** peut avoir **plusieurs Produits** (si la collection existe dans `Categorie`)

📌 Pourquoi ça fonctionne ?

EF suit une convention appelée "Foreign Key Discovery" :

- Si tu as une propriété `NomEntitéId` (ex : `CategorieId`)
- Et une propriété de navigation `NomEntité` (ex : `Categorie`)

→ EF relie les deux automatiquement.

💡 Tu peux aller plus loin (optionnel)


Dans `Categorie.cs`, tu peux ajouter la collection inverse :

C#

```
1 public List<Produit> Produits { get; set; }
2
```

Cela rend la relation **bidirectionnelle** :

- Un `Produit` connaît sa `Categorie`
- Une `Categorie` connaît tous ses `Produits`

 **Bonus : tu peux le forcer avec Fluent API si nécessaire**

C#

```
1 modelBuilder.Entity<Produit>()
2     .HasOne(p => p.Categorie)
3     .WithMany(c => c.Produits)
4     .HasForeignKey(p => p.CategorieId);
5
```

Mais dans 90 % des cas, les conventions suffisent et permettent un code plus simple et plus lisible.

✓ Résumé des conventions de clé étrangère implicite dans EF Core

Propriété de navigation	Nom de propriété acceptée comme FK	Commentaire
<code>Categorie</code>	<code>CategorieId</code>	✓ Le plus courant : NomEntité + Id
<code>Categorie</code>	<code>IdCategorie</code>	✓ Accepté aussi, surtout en français
<code>Categorie</code>	<code>Categorie_Id</code>	✓ Compatible (ancienne convention EF6)
<code>Categorie</code>	<code>CategorieID</code> (avec majuscule)	✓ Toléré (pas recommandé par convention)
<code>Categorie</code>	<code>CategorieKey</code>	✗ Non reconnu automatiquement
<code>Categorie</code>	<code>FkCategorie</code>	✗ À configurer manuellement en Fluent API

✓ Principales conventions automatiques d'Entity Framework Core

Comportement	Convention
Clé primaire (PK)	Une propriété nommée <code>Id</code> ou <code>NomDeClasseId</code> est automatiquement considérée comme la clé primaire
Clé étrangère (FK)	Une propriété nommée <code>NomNavigationId</code> (ex: <code>CategorieId</code>) liée à une navigation <code>Categorie</code>
One-to-many	Si une entité <code>Produit</code> a une <code>Categorie</code> et <code>Categorie</code> a une collection <code>Produits</code> , EF infère <code>one-to-many</code>
One-to-one	Deux propriétés de navigation pointant mutuellement l'une vers l'autre sans collection
Nom de table	Le nom de la classe devient le nom de la table (ex : <code>Produit</code> → <code>Produit</code>)
Nom de colonne	Le nom de la propriété devient le nom de la colonne
Type de colonne	Le type C# est automatiquement mappé au type SQL (ex: <code>string</code> →

	<code>nvarchar(max)</code>
Nullable / Not Null	Les types nullable (<code>string?</code> , <code>int?</code>) deviennent <code>NULL</code> en SQL, les autres sont <code>NOT NULL</code>
Cascade Delete	Activé automatiquement pour les relations <code>required</code> avec clé étrangère
Index implicite sur PK et FK	EF crée automatiquement un index sur les PK et FK

Tuto 20/05 Seed

Initialiser une base Code First (Seed + Outils)

1. Seed de données de base

Dans `CosmosModels`, tu peux ajouter des données par défaut via `OnModelCreating` dans le `DbContext`.

```

1 protected override void OnModelCreating(ModelBuilder modelBuilder)
2 {
3     modelBuilder.Entity<Categorie>().HasData(
4         new Categorie { Id = 1, Nom = "Papeterie" },
5         new Categorie { Id = 2, Nom = "Informatique" }
6     );
7
8     modelBuilder.Entity<Produit>().HasData(
9         new Produit { Id = 1, Nom = "Clavier", Prix = 25.99m, CategorieId = 2 },
10        new Produit { Id = 2, Nom = "Cahier", Prix = 2.50m, CategorieId = 1 }
11    );
12 }
13

```

2. Lien avec les migrations EF Core

- `HasData()` fonctionne uniquement via une migration.
- Toute modification dans le seed nécessite une nouvelle migration.
- EF stocke l'état du seed dans `ModelSnapshot.cs`.

3. Gestion dans la migration

EF génère automatiquement du code comme :

```

1 migrationBuilder.InsertData(
2     table: "Categorie",
3     columns: new[] { "Id", "Nom" },
4     values: new object[] { 1, "Papeterie" }
5 );
6

```

4. Réinitialiser la base (en dev)

1. Supprimer la base depuis SSMS
2. Refaire un `Update-Database` :

PowerShell

```
1 Update-Database -Project CosmosModels
2
```

5. Afficher les données seedées

Clic droit => Ajouter contrôleur avec Entity Framework

Vue **Index.cshtml** :

```
1 @model IEnumerable<Categorie>
2
3 <h2>Catégories disponibles</h2>
4 <ul>
5 @foreach (var cat in Model)
6 {
7     <li>@cat.Nom</li>
8 }
9 </ul>
10
```

HTML

Contrôleur **CategorieController** :

```
1 public class CategorieController : Controller
2 {
3     private readonly CosmosLightContext _context;
4
5     public CategorieController(CosmosLightContext context)
6     {
7         _context = context;
8     }
9
10    public IActionResult Index()
11    {
12        return View(_context.Categories.ToList());
13    }
14 }
15
```

C#

Tuto 27/05 Formulaire

Petite explication de la correction apportée dans Produit.cs

```
1 public int CategorieId { get; set; }
2 [ValidateNever]
3 public Categorie Categorie
4 {
5     get; set;
6 }
7 }
8
```

C#

[ValidateNever] est pour indiquer à MVC que le type ne rentre pas dans les validations formulaire, le besoin de ViewModel dans certains cas restent valident

 **Objectif :**

Améliorer la page 'Index' de 'Produit' pour :

Améliorer la page `Index` de `Produit` pour :

- Faire une recherche par nom
- Trier par nom ou prix
- Filtrer par catégorie

✅ Étape 1 – Ajouter les paramètres à l'action Index

Dans `ProduitController.cs` :

C#

```
1 public async Task<IActionResult> Index(string search, string sort, int? categorieId)
2 {
3     var produits = _context.Produits.Include(p => p.Categorie).AsQueryable();
4
5     if (!string.IsNullOrWhiteSpace(search))
6         produits = produits.Where(p => p.Nom.Contains(search));
7
8     if (categorieId != null)
9         produits = produits.Where(p => p.CategorieId == categorieId);
10
11     produits = sort switch
12     {
13         "prix" => produits.OrderBy(p => p.Prix),
14         "prix_desc" => produits.OrderByDescending(p => p.Prix),
15         "nom_desc" => produits.OrderByDescending(p => p.Nom),
16         _ => produits.OrderBy(p => p.Nom)
17     };
18
19     ViewBag.CurrentSearch = search;
20     ViewBag.CurrentSort = sort;
21     ViewBag.CategorieId = categorieId;
22     ViewBag.Categories = new SelectList(await _context.Categories.ToListAsync(), "Id", "Nom");
23
24     return View(await produits.ToListAsync());
25 }
26
```

✅ Étape 2 – Vue Index.cshtml

En haut de la page, ajoute un **formulaire de recherche et de filtre** :

HTML

```
1 <form method="get" class="form-inline mb-3">
2     <input type="text" name="search" value="@ViewBag.CurrentSearch" placeholder="Rechercher..."
3     class="form-control mr-2" />
4     <select name="categorieId" class="form-control mr-2">
5         <option value="">Toutes les catégories</option>
6         @foreach (var cat in ViewBag.Categories as SelectList)
7         {
8             <option value="@cat.Value" @(ViewBag.CategorieId?.ToString() == cat.Value ?
9             "selected" : "")>@cat.Text</option>
10        }
11    </select>
12    <button type="submit" class="btn btn-primary">Filtrer</button>
13 </form>
14
```

✅ Étape 3 – Ajouter des liens de tri dans le tableau

HTML

```
1 <table class="table">
```

```

1 <thead>
2 <tr>
3 <th>
4 <a asp-action="Index" asp-route-sort="@(<ViewBag.CurrentSort == "nom_desc" ? "" :
"nom_desc")">Nom</a>
5 </th>
6 <th>
7 <a asp-action="Index" asp-route-sort="@(<ViewBag.CurrentSort == "prix_desc" ?
"prix" : "prix_desc")">Prix</a>
8 </th>
9 <th>Catégorie</th>
10 </tr>
11 </thead>
12 <tbody>
13 @foreach (var p in Model)
14 {
15 <tr>
16 <td>@p.Nom</td>
17 <td>@p.Prix.ToString("0.00")</td>
18 <td>@p.Categorie?.Nom</td>
19 </tr>
20 }
21 </tbody>
22 </table>
23
24

```

Astuce : conserver les filtres dans les liens de tri

Pour garder `search` et `categorieId` :

```

1 <a asp-action="Index"
2   asp-route-sort="nom"
3   asp-route-search="@ViewBag.CurrentSearch"
4   asp-route-categorieId="@ViewBag.CategorieId">Nom</a>
5

```

HTML

Résultat attendu :

- Un champ de recherche filtre les noms
- Un menu déroulant filtre par catégorie
- Les colonnes Nom et Prix sont cliquables pour trier croissant / décroissant

Tuto 03/06 ModelState

Objectifs :

- Afficher un message si aucun résultat n'est trouvé
- Utiliser `TempData` pour transmettre un message après une action
- Gérer les erreurs comme `NotFound`, `BadRequest`
- Structurer proprement les messages dans la vue

1. Afficher un message si aucun élément

Dans la vue `Index.cshtml`, remplace la section `@foreach` par :

```

1 @if (!Model.Any())
2 {

```

HTML

```

3      <tr>
4          <td class="alert alert-info" colspan="3">
5              Aucun produit ne correspond à votre recherche.
6          </td>
7      </tr>
8  }
9  else
10 {
11     @foreach (var p in Model)
12     {
13         <tr>
14             <td>@p.Nom</td>
15             <td>@p.Prix.ToString("0.00")</td>
16             <td>@p.Categorie?.Nom</td>
17         </tr>
18     }
19 }

```

Mini-débat, souvent des outils comme Visual Studio peuvent suggérer `Count() > 0` par rapport à `Any()`, donc voici comment lui faire fermer sa gueule

✓ Any()

- **But** : Vérifie s'il existe **au moins un élément**.
- **Optimisé** : Dès qu'un élément est trouvé, la méthode **s'arrête** (court-circuit).
- **Performant** : Particulièrement efficace sur des collections **IEnumerable** ou des requêtes LINQ vers une **base de données** (comme Entity Framework).
- **SQL généré (EF)** : `SELECT TOP(1) ...`, donc plus léger.

⚠ Count() > 0

- **But** : Compte **tous les éléments** puis compare le total.
- **Moins performant** : Même si un seul élément suffit à répondre à la condition, la méthode va **tout parcourir / compter**.
- **SQL généré (EF)** : `SELECT COUNT(*) ...`, donc plus **lourd**, surtout sur de grandes tables.

✓ 2. Utiliser TempData ou ViewBag pour afficher un message après action

Dans le contrôleur (DeleteConfirmed, Create, etc.) :

C#

```

1 TempData["Message"] = "Le produit a bien été supprimé.";
2 ViewBag.Message = $"Le produit \"{produit.Nom}\" a bien été supprimé.";
3 return RedirectToAction(nameof(Index));

```

Dans la vue Index.cshtml (tout en haut) :

HTML

```

1 @if (TempData["Message"] is string msg)
2 {
3     <div class="alert alert-success">
4         @msg
5     </div>
6 }
7
8 @if (ViewBag.Message != null)
9 {
10     <div class="alert alert-success">@ViewBag.Message</div>
11 }

```

→ TempData dure une requête, donc parfait pour les redirections après POST.

✓ 3. Gérer proprement NotFound et BadRequest

Dans Edit, Details, etc. :

```

1 public async Task<IActionResult> Details(int? id)
2 {
3     if (id == null)
4         return BadRequest(); // 400 Bad Request
5
6     var produit = await _context.Produits
7         .Include(p => p.Categorie)
8         .FirstOrDefaultAsync(p => p.Id == id);
9
10    if (produit == null)
11        return NotFound(); // 404 Not Found
12
13    return View(produit);
14 }
15

```

→ BadRequest() → requête invalide (paramètre manquant, etc.)

→ NotFound() → ressource absente (ID non existant)

✓ 4. Affichage conditionnel plus fin

Tu peux afficher dynamiquement un bloc d'information selon des critères de ton ViewModel, de l'utilisateur connecté, etc. :

```

1 @if (User.IsInRole("Admin"))
2 {
3     <a href="/Produit/Create" class="btn btn-primary">Ajouter un produit</a>
4 }

```

Ou selon une propriété dans le ViewModel :

```

1 @if (Model.ShowAdvancedSearch)
2 {
3     <div class="card card-body">Recherche avancée</div>
4 }

```

Tu peux aussi inclure un champ Message directement dans un ViewModel, par exemple :

```

1 public class ProduitIndexViewModel
2 {
3     public string? Message { get; set; }
4     public List<Produit> Produits { get; set; } = new();
5 }

```

```

1 var model = new ProduitIndexViewModel
2 {
3     Produits = produits.ToList(),
4     Message = "Suppression réussie"
5 }

```

```
5 };
6 return View(model);
```

HTML

```
1 @if (!string.IsNullOrEmpty(Model.Message))
2 {
3     <div class="alert alert-success">@Model.Message</div>
4 }
```

Résumé

	≡ Cas	≡ Outil / méthode
1	Aucun résultat	<code>if (!Model.Any())</code>
2	Message après action	<code>TempData["Message"]</code>
3	Paramètre manquant	<code>BadRequest()</code>
4	Élément non trouvé	<code>NotFound()</code>
5	Affichage dynamique	<code>@if (...) {}</code>

✓ 5. Validation des données avec ModelState.IsValid

Quand tu postes un formulaire (ex : `Create`, `Edit`), ASP.NET Core vérifie automatiquement les **annotations de validation** (`[Required]`, `[Range]`, etc.).

Le résultat est accessible via la propriété :

C#

```
1 if (!ModelState.IsValid)
2 {
3     // Formulaire invalide → on renvoie la vue avec le modèle
4     return View(viewModel);
5 }
6
```

Exemple complet dans Create :

C#

```
1 [HttpPost]
2 [ValidateAntiForgeryToken]
3 public async Task<IActionResult> Create(ProduitFormViewModel viewModel)
4 {
5     if (!ModelState.IsValid)
6     {
7         // On recharge la liste déroulante pour ne pas la perdre
8         viewModel.Categories = await _context.Categories
9             .Select(c => new SelectListItem { Value = c.Id.ToString(), Text = c.Nom })
10            .ToListAsync();
11
12         return View(viewModel);
13     }
14
15     _context.Produits.Add(viewModel.Produit);
```

```
16     await _context.SaveChangesAsync();
17
18     TempData["Message"] = "Produit créé avec succès.";
19     return RedirectToAction(nameof(Index));
20 }
21
```

En lien avec la vue :

- `asp-validation-for="..."` va afficher les erreurs liées à chaque champ
- `asp-validation-summary` peut afficher tous les messages regroupés

	≡ Avantage	≡ Détail
1	✓ Séparation claire	C'est le contrôleur qui décide si la donnée est correcte, pas la vue
2	✓ Plus sécurisant	Empêche de sauvegarder des données corrompues
3	✓ UX robuste	Affiche des erreurs claires sans recharger la page

Pour finir avant QCM, voici une proposition de parents pour toutes les entités de l'application

```
[NotMapped]
public class EntityId
{
    public int Id { get; set; }
}

[NotMapped]
public class EntityIdAndName
{
    public int Id { get; set; }
    public string Name { get; set; }
}

[NotMapped]
public class EntityGuid
{
    public Guid Id { get; set; }
}

[NotMapped]
public class EntityGuidAndName
{
    public Guid Id { get; set; }
    public string Name { get; set; }
}
```

Ainsi l'entité Produit devient ceci

```
public class Produit : EntityIdAndName
{
    public decimal Prix { get; set; }
}
```

Si temps y'a un début de plan pour faire CosmosV5 de commencé si besoin

Tuto 10/06 Suppression/Archivage d'un produit

Objectif :

- Ajouter un champ Actif à l'entité Produit
- Masquer les produits inactifs de l'Index
- Ajouter une action "Archiver" dans le contrôleur
- Ajouter un filtre pour afficher les produits archivés si besoin

Étape 1 – Ajouter le champ dans Produit.cs

C#

```
1 public bool Actif { get; set; } = true;
```

On pourrait aller avec l'idée des parents EntityId et mettre une interface pour imposer le champ aux entités

```
public interface IArchivable
{
    bool Actif { get; set; }
}
```

Pour quelque chose du genre

```
public class Produit : EntityIdAndName, IArchivable
{
    public decimal Prix { get; set; }
    public bool Actif { get; set; } = true;

    public int CategorieId { get; set; }
    public Categorie? Categorie { get; set; }
}
```

On pourrait aller plus loin avec des ArchivableExtensions et ArchivableQueryExtensions, peut-être un futur tuto ?

Mais je disgresse ... SGRESSE

Étape 2 – Nouvelle migration

JavaScript

```
1 Add-Migration AddActifToProduit
2 Update-Database
```

Étape 3 – Modifier Index dans ProduitsController.cs

Ajoute un booléen showArchived :

C#

```
1 public async Task<IActionResult> Index(string search, string sort, int? categorieId, bool
  showArchived = false)
2 {
3     var produits = _context.Produits.Include(p => p.Categorie).AsQueryable();
4
```

```

5     if (!showArchived)
6         produits = produits.Where(p => p.Actif);
7
8     if (!string.IsNullOrWhiteSpace(search))
9         produits = produits.Where(p => p.Nom.Contains(search));
10
11    if (categorieId != null)
12        produits = produits.Where(p => p.CategorieId == categorieId);
13
14    produits = sort switch
15    {
16        "prix" => produits.OrderBy(p => p.Prix),
17        "prix_desc" => produits.OrderByDescending(p => p.Prix),
18        "nom_desc" => produits.OrderByDescending(p => p.Nom),
19        _ => produits.OrderBy(p => p.Nom)
20    };
21
22    ViewBag.CurrentSearch = search;
23    ViewBag.CurrentSort = sort;
24    ViewBag.CategorieId = categorieId;
25    ViewBag.Categories = new SelectList(_context.Categories.ToList(), "Id", "Nom",
ViewBag.CategorieId);
26    ViewBag.ShowArchived = showArchived;
27
28    return View(await produits.ToListAsync());
29 }

```

Étape 4 – Lien "Archiver" dans la vue Index.cshtml

Dans le tableau :

```

1 @if (p.Actif)
2 {
3     <a asp-action="Archive" asp-route-id="@p.Id" class="btn btn-warning btn-sm">Archiver</a>
4 }
5 else
6 {
7     <span class="text-muted">Archivé</span>
8 }

```

Étape 5 – Nouvelle action dans ProduitsController.cs

```

1 public async Task<IActionResult> Archive(int id)
2 {
3     var produit = await _context.Produits.FindAsync(id);
4     if (produit == null)
5         return NotFound();
6
7     produit.Actif = false;
8     _context.Update(produit);
9     await _context.SaveChangesAsync();
10
11    return RedirectToAction(nameof(Index));
12 }

```

Étape 6 – Afficher les produits archivés

En haut de la vue Index.cshtml, ajoute un lien ou une case à cocher :

```

1 <a asp-action="Index" asp-route-showArchived="true" class="btn btn-outline-secondary">
2     Voir les archivés

```

3

✓ Résultat attendu

- Les produits sont visibles uniquement s'ils sont `Actif == true`
- Un bouton "Archiver" les rend invisibles par défaut
- On peut voir les archivés en filtrant

Tuto 17/06 Extensions

Pour répondre à la question sur l'enchaînement des requêtes LINQ et les performances :

Tant qu'on reste dans du IQueryable, tout est en "deferred execution", donc tant qu'on ne renvoie pas une List ou une entité (avec .ToListAsync(), .FirstOrDefaultAsync(), .CountAsync(), etc) rien n'est exécuté en base. C'est Entity Framework qui prend les Where et OrderBy et fait une seule requête à la fin quand on lui demande la liste.

Pour reprendre ce qui a été dit au dernier tuto

On pourrait aller avec l'idée des parents `EntityId` et mettre une interface pour imposer le champ aux entités

```
public interface IArchivable
{
    bool Actif { get; set; }
}
```

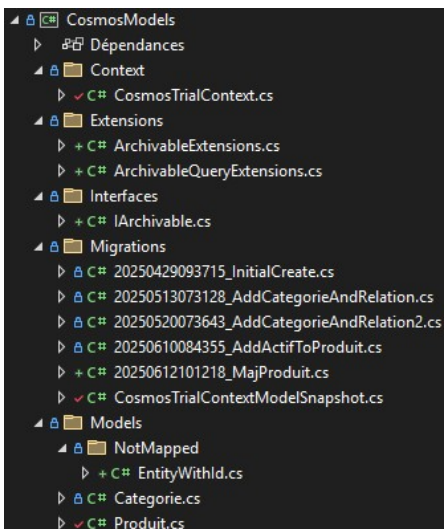
Pour quelque chose du genre

```
public class Produit : EntityIdAndName, IArchivable
{
    public decimal Prix { get; set; }
    public bool Actif { get; set; } = true;

    public int CategorieId { get; set; }
    public Categorie? Categorie { get; set; }
}
```

On pourrait aller plus loin avec des `ArchivableExtensions` et `ArchivableQueryExtensions`, peut-être un futur tuto ?

Donc dans CosmosTrial, Produit hérite de EntityWithIdAndName et implémente IArchivable et les extensions sont créées ce qui donne dans l'arborescence de CosmosModels



```
namespace CosmosModels.Models.NotMapped
{
    [NotMapped]
    1 référence | 0 modification | 0 auteur, 0 modification
    public class EntityWithId
    {
        [Key]
        11 références | 0 modification | 0 auteur, 0 modification
        public int Id { get; set; }
    }

    [NotMapped]
    1 référence | 0 modification | 0 auteur, 0 modification
    public class EntityWithIdAndName : EntityWithId
    {
        [Required]
        16 références | 0 modification | 0 auteur, 0 modification
        public required string Name { get; set; }
    }
}
```

Vu que j'ai mis en anglais, Produit a été mis à jour avec la migration suivante, d'ailleurs vu qu'on "franglaise", faudrait peut-être se faire un CosmosV5 - Development Guidelines ?

```
namespace CosmosModels.Migrations
{
    /// <inheritdoc />
    1 référence | 0 modification | 0 auteur, 0 modification
    public partial class MajProduit : Migration
    {
        /// <inheritdoc />
        0 références | 0 modification | 0 auteur, 0 modification
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.RenameColumn(
                name: "Nom",
                table: "Produits",
                newName: "Name");
        }

        /// <inheritdoc />
        0 références | 0 modification | 0 auteur, 0 modification
        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.RenameColumn(
                name: "Name",
                table: "Produits",
                newName: "Nom");
        }
    }
}
```

Produit est un poil plus simple

```
public class Produit : EntityWithIdAndName, IArchivable
{
    [Range(0.01, 10000)]
    15 références | Gwenael LE PAGE, il y a 44 jours | 1 auteur, 1 modification
    public decimal Prix { get; set; }

    6 références | Gwenael LE PAGE, il y a 2 jours | 1 auteur, 1 modification
    public bool Actif { get; set; } = true;

    12 références | Gwenael LE PAGE, il y a 30 jours | 1 auteur, 1 modification
    public int CategorieId { get; set; }

    [ValidateNever]
    8 références | Laury Poyer, il y a 19 jours | 2 auteurs, 2 modifications
    public Categorie Categorie
    {
        get; set;
    }
}
```

Qu'est-ce qu'une méthode d'extension ?

Une méthode d'extension est une méthode statique qui "ajoute" une méthode à un type, sans modifier ce type.

Exemple typique :

C#

```

1 public static class StringExtensions
2 {
3     public static bool IsNullOrWhiteSpace(this string input)
4     {
5         return string.IsNullOrWhiteSpace(input);
6     }
7 }

```

"".IsNullOrWhiteSpace(); // fonctionne même si string n'a pas cette méthode à l'origine

```

public static class ArchivableExtensions
{
    public static void Archiver(this IArchivable entity)
    {
        entity.Actif = false;
    }
}

```

→ Ici, le `this IArchivable` dans la signature indique à C# :

"si un objet implémente `IArchivable`, autorise les appels `.Archiver()` dessus".

Il suffit donc que ton entité implémente bien `IArchivable` et que ton fichier `ArchivableExtensions.cs` soit importé via `using`, par exemple :

```

csharp
using CosmosTrial.Extensions; // ou Le bon namespace
produit.Archiver();

```

Fichier	Cible des extensions	Exemple	Type de logique
ArchivableExtensions.cs	IArchivable (objet unique)	produit.Archiver();	Manipulation
ArchivableQueryExtensions.cs	IQueryable<T> (requête EF)	.WhereActif()	Filtrage dans LINQ

✓ Pourquoi deux fichiers ?

C'est la convention C# recommandée :

- Les extensions qui agissent sur un objet isolé vont dans un fichier ciblé sur le type (`IArchivable`)
- Les extensions qui modifient une requête LINQ (`IQueryable<T>`) sont séparées, car elles :
 - sont génériques (`T : IArchivable`)
 - n'ont pas les mêmes `using`
 - agissent en amont d'un `ToListAsync()` → impact SQL

Voici les fichiers Archivable

```

namespace CosmosModels.Extensions
{
    0 références | 0 modification | 0 auteur, 0 modification
    public static class ArchivableExtensions

```

```

public static class ArchivableExtensions
{
    1 référence | 0 modification | 0 auteur, 0 modification
    public static void Archiver(this IArchivable entity)
    {
        entity.Actif = false;
    }

    0 références | 0 modification | 0 auteur, 0 modification
    public static void Restaurer(this IArchivable entity)
    {
        entity.Actif = true;
    }
}

```

```

namespace CosmosModels.Extensions
{
    0 références | 0 modification | 0 auteur, 0 modification
    public static class ArchivableQueryExtensions
    {
        1 référence | 0 modification | 0 auteur, 0 modification
        public static IQueryable<T> WhereActif<T>(this IQueryable<T> source) where T : class, IArchivable => source.Where(e => e.Actif);

        1 référence | 0 modification | 0 auteur, 0 modification
        public static IQueryable<T> WhereArchived<T>(this IQueryable<T> source) where T : class, IArchivable => source.Where(e => !e.Actif);
    }
}

```

Voici les utilisations dans le contrôleur

```

0 références | Gwenaél LE PAGE, il y a 2 jours | 1 auteur, 1 modification
public async Task<IActionResult> Archive(int id)
{
    var produit = await _context.Produits.FindAsync(id);
    if (produit == null)
        return NotFound();

    produit.Actif = false;
    produit.A
    _context
    await _c
    return R
}

```

Actif
Archiver CosmosModels.Extensions
Categorie
CategorieId
Equals
GetHashCode
GetType
Id
Nom

(Extension) void CosmosModels.Interfaces.IArchivable.Archiver()

```

produits = showArchived ? produits.WhereArchived() : produits.WhereActif();

//Devient inutile
//if (!showArchived)
//    produits = produits.Where(p => p.Actif);

```

Et aussi petite mise à jour du filtrage sur la combo categories

```

var categories = _context.Categories.ToList();
categories.Insert(0, new Categorie { Id = 0, Nom = "Toutes" });
ViewBag.Categories = new SelectList(categories, "Id", "Nom", categorieId ?? 0);

```

D'autres idées d'extensions

```

public interface IAuditable
{
    string? CreePar { get; set; }
    DateTime CreeLe { get; set; }
    string? ModifiePar { get; set; }
    DateTime? ModifieLe { get; set; }
}

```



```
public static class AuditableExtensions
{
    public static void SetCreated(this IAuditable entity, string user)
    {
        entity.CreePar = user;
        entity.CreeLe = DateTime.UtcNow;
    }

    public static void SetModified(this IAuditable entity, string user)
    {
        entity.ModifiePar = user;
        entity.ModifieLe = DateTime.UtcNow;
    }
}
```

- ◆ **Extension générique :**

```
csharp  Copier  Modifier
```

```
public static class PaginationExtensions
{
    public static IQueryable<T> Paginate<T>(this IQueryable<T> query, int page, int pageSize)
    {
        return query.Skip((page - 1) * pageSize).Take(pageSize);
    }
}
```

➡ Usage :

```
csharp
```

Copier

Modifier

```
var pageProduits = _context.Produits.Paginate(1, 10).ToList();
```

➡ Bonus : compatible avec toutes les entités, aucun `interface` nécessaire

Notez bien que là l'objet c'est IQueryable.

Pour terminer voici un cas où ce n'est pas un IQueryable<T>

```
public static class UserQueryExtensions
{
    public static IQueryable<User> Search(this IQueryable<User> query, string? keyword)
    {
        if (string.IsNullOrEmpty(keyword))
            return query;

        keyword = keyword.ToLower();

        return query.Where(u =>
            u.Nom.ToLower().Contains(keyword) ||
            u.Prenom.ToLower().Contains(keyword));
    }
}
```

Voici la même mais façon LIKE SQL

```
public static IQueryable<Utilisateur> Search(this IQueryable<Utilisateur> query, string? keyword)
{
    if (string.IsNullOrEmpty(keyword))
        return query;

    return query.Where(u =>
        EF.Functions.Like(u.Nom, $"%{keyword}%") ||
        EF.Functions.Like(u.Prenom, $"%{keyword}%"));
```

```
}  
    }  
}
```

Tuto 24/06 Migration

Cas 1 : Mauvais nom ou migration inutile

Ajouter un champ bidon dans Produit

Add-Migration CeNomDeMigrationEstPourri

Finalement ça ne convient pas et on n'a pas fait Update-Database

Solution

Remove-Migration

Cas 2 : Migration appliquée mais contenant une erreur

Scénario

Ajouter un champ bidon dans Produit (déjà fait dans Cas 1)

Add-Migration ChampBidon

Update-Database

Étapes :

1. Annuler la migration dans la base

Update-Database NomDeLaMigrationPrécédente donc normalement Update-Database MajProduit

2. Supprimer le fichier de migration en trop

Remove-Migration

3. Corriger le modèle (ou pas ...)

4. Créer une nouvelle migration propre

Add-Migration ChampBidonMaisMieux

Update-Database

Cas 3 : Repartir de zéro en local avec une base vide (reset de dev)

Scénario

- Supprimer toutes les données
- Rejouer toutes les migrations
- Garder l'historique Git propre (pas supprimer les migrations)

Solution

1. Supprimer la base de données localement

```
1 Drop-Database
```

Shell

Recréer la base avec toutes les migrations

Shell

1 Update-Database

Variantes utiles

- Si tu veux revenir à une migration précise :

Shell

1 Update-Database MigrationAvantBug

- Si tu veux tester un Seed complet (cf. tuto précédent), ce reset te permet de rejouer les HasData()

Pourquoi c'est important

	≡ Besoin	≡ Solution
1	Base test cassée / incohérente	Drop-Database + Update-Database
2	Rejouer un seed	Même méthode
3	Faire une démo propre à un collègue	Idéal pour simuler un "first run"

Bonus – Fichiers clés

	≡ Fichier	≡ Rôle
1	__EFMigrationsHistory	Table SQL qui liste toutes les migrations appliquées
2	YourDbContextModelSnapshot.cs	Représente l'état final connu du modèle (sert de base de diff)
3	Dossier Migrations/	Contient chaque fichier .cs de migration générée

QCM

QCM 1

Tu fais Add-Migration TestMigration par erreur, sans avoir modifié les modèles. Que faire si tu n'as pas encore fait Update-Database ?

- A. Supprimer le fichier manuellement
- B. Faire `Remove-Migration`
- C. Faire `Update-Database` quand même
- D. Tout recommencer depuis la base de données

? QCM 2

Tu as fait `Update-Database` mais tu veux annuler la dernière migration car elle contient une erreur. Quelle est la bonne séquence ?

- A. Supprimer le fichier puis faire `Update-Database`
- B. Faire `Remove-Migration` puis `Update-Database`
- C. Faire `Update-Database` vers l'ancienne migration puis `Remove-Migration`
- D. Tu ne peux plus rien faire

? QCM 3

À quoi sert le fichier `YourDbContextModelSnapshot.cs` ?

- A. Il stocke les données de test
- B. Il représente le modèle EF tel qu'il est appliqué à la base
- C. Il enregistre les erreurs de migration
- D. Il contient le nom de toutes les entités de ton projet

Les réponses sont visibles depuis mon bureau :

P-O-R-S-C-H-E ?

Tuto 01/07 Migration en PROD

Cas classique chez un client

Pas de "Update-Database" en prod, car :

- Pas d'accès au terminal en général
- Si le client veut valider (surtout si DBA)
- Il faut un SQL lisible / archivable / traçable (surtout si DBA)

Étapes recommandées

1. Générer le SQL de migration côté dev

```
1 Script-Migration -From NomMigrationPrécédente -To NomMigrationActuelle -Output Migrations/
  Migration_2024_06_28.sql
```

Exemple :

```
1 Script-Migration -From MajProduit -To AddChampDescription -Output Migrations/Migration_Desc.sql
```

♦ Ce script contient :

- Les ALTER TABLE, ADD COLUMN, etc.
- Un __EFMigrationsHistory update à la fin
- Traçable et versionnable
- Gérer les particularités clients ou SGBD

2. Vérifier / nettoyer le script SQL généré

- Supprimer les éventuels `GO` ou `PRINT` selon la cible (SSMS, Azure, etc.)
- Ajouter un BEGIN TRANSACTION / COMMIT si souhaité
- Ajouter un commentaire de version, ex :

```
1 -- Migration EF : AddChampDescription (CosmosTrial - v1.2.4)
```

SQL

3. Transmettre et appliquer côté client

- Le client exécute le SQL avec son outil habituel (SSMS, pgAdmin, etc.)
- Ou un script automatique l'applique dans un pipeline de déploiement

Bonus : éviter les pièges

	≡ Mauvaise pratique	≡ À éviter pourquoi ?
1	Faire <code>Update-Database</code> direct en prod	Risque d'incompatibilité, requêtes non tracées
2	Générer un script trop général	Il faut toujours spécifier <code>-From</code> / <code>-To</code>
3	Modifier manuellement une migration EF	Cela casse le snapshot et la cohérence EF

Cas 1 – Migration EF déjà appliquée partiellement par accident

Scénario

Un dev a fait un `Update-Database` en prod ou en recette, mais :

- la migration a échoué en plein milieu
- le fichier `.cs` est modifié ou supprimé depuis

Solution recommandée

1. Vérifier la base : la table `__EFMigrationsHistory` contient-elle la migration fautive ?
2. Rejouer la migration avec un script SQL généré en forçant `-Idempotent` :

Shell

```
1 Script-Migration -To NomMigration -Idempotent -Output Fix_Migration.sql
```

 `-Idempotent` permet d'avoir un script qui ne casse pas si la migration a été appliquée partiellement.

Cas 2 – Base de données Azure SQL

Particularité :

- `GO` n'est **pas accepté** par défaut dans Azure SQL exécuté via `SqlCommand` ou API .NET
- Les scripts doivent être **compatibles Azure**

Solution :

1. Générer un script SQL **sans `GO`** ou **compatible Azure** :
 - Utiliser Visual Studio pour l'export SQL (avec `Advanced Save Options`)
 - Ou parser / splitter manuellement le SQL
2. Vérifier le timeout et les transactions

Cas 3 – Multi-migrations à appliquer (le client a du retard)

Scénario

Le client n'a pas appliqué les 5 dernières migrations.

Solution :

```
1 Script-Migration -From NomDernièreMigrationAppliquée -To NomDernière -Output  
Update_Client_v1_2_3_to_v1_3_0.sql
```

Shell

- Sans utiliser le `-From` de `Script-Migration`, il prendra depuis le début
- Éventuellement ajouter des commentaires en tête de chaque bloc pour **segmenter les versions**

Cas 4 – Rollback de migration EF Core

Scénario


Une migration a été appliquée en prod... mais elle casse l'application ou a introduit un bug métier. Il faut **revenir en arrière**.

Solution correcte (rollback réel)

1. On créer le script de migration dans l'autre sens !

```
1 Script-Migration -From v1.3 -To v1.2 -Output Rollback_ChampBidon.sql  
2 Update-Database NomMigrationPrécédente
```

Shell

 L'ordre est inversé ici (`-From` = actuelle, `-To` = cible précédente)

1. Vérifier le contenu

- Le script contient les instructions `DROP COLUMN`, `DROP CONSTRAINT`, etc.
- En toute fin, EF supprimera la ligne de `__EFMigrationsHistory` correspondante

2. Transmettre le script SQL au client

3. Priez ! (je parle de la compétence du client, pas du script)

Cas 5 – Gestion SQL Server et PostgreSQL côté dev

En gros le switch entre Sql Server et PostgreSQL se ferait avec Program.cs et appsettings.json (il est possible d'avoir les 2 DbContext en même temps et que ce soit le service qui renvoi avec le bon Context, un peu ce que faisait CosmosMVC version database first avec par exemple Service.GetAll() dans les Contrôleurs mais ça reste une solution assez lourde)

```
var builder = WebApplication.CreateBuilder(args);

// On lit une seule clé de connexion, qui peut pointer vers SQL Server ou PostgreSQL
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection")
    ?? throw new Exception("Connection string manquante");

// On détecte Le provider à partir du contenu de la chaîne de connexion
if (connectionString.Contains("Server=", StringComparison.OrdinalIgnoreCase))
{
    builder.Services.AddDbContext<AppDbContext>(options =>
        options.UseSqlServer(connectionString));
}
else if (connectionString.Contains("Host=", StringComparison.OrdinalIgnoreCase))
{
    builder.Services.AddDbContext<AppDbContext>(options =>
        options.UseNpgsql(connectionString));
}
else
{
    throw new Exception("Base de données non supportée ou chaîne de connexion invalide.");
}
```

Ou un truc plus fin

```
json

"Database": {
  "Provider": "SqlServer",
  "ConnectionString": "Server=localhost;Database=Cosmos;Trusted_Connection=True"
}
```

Puis :

```
csharp

var provider = builder.Configuration["Database:Provider"];
var connectionString = builder.Configuration["Database:ConnectionString"];

if (provider == "SqlServer")
    builder.Services.AddDbContext<AppDbContext>(o => o.UseSqlServer(connectionString));
else if (provider == "PostgreSql")
    builder.Services.AddDbContext<AppDbContext>(o => o.UseNpgsql(connectionString));
else
    throw new Exception("Provider inconnu");
```

Ce genre de cas serait même possible

appsettings.Development.json

json

[Copier](#) [Modifier](#)

```
{
  "Database": {
    "Provider": "SqlServer",
    "ConnectionString": "Server=localhost;Database=CosmosDev;Trusted_Connection=True;TrustServerCertificate=True;"
  }
}
```

appsettings.Production.json

json

[Copier](#) [Modifier](#)

```
{
  "Database": {
    "Provider": "PostgreSql",
    "ConnectionString": "Host=prod-db.host.internal;Database=cosmos;Username=admin;Password=motdepasse;"
  }
}
```

Mais c'est ainsi que ça va nous intéresser

✓ Exemple de appsettings.SqlServer.json

json

[Copier](#) [Modifier](#)

```
{
  "Database": {
    "Provider": "SqlServer",
    "ConnectionString": "Server=localhost;Database=CosmosDev;Trusted_Connection=True;TrustServerCertificate=True;"
  }
}
```

✓ Exemple de appsettings.PostgreSQL.json

json

[Copier](#) [Modifier](#)

```
{
  "Database": {
    "Provider": "PostgreSQL",
    "ConnectionString": "Host=localhost;Database=cosmos;Username=postgres;Password=pass"
  }
}
```

► Générer une migration SQL Server

powershell

[Copier](#) [Modifier](#)

```
$env:DOTNET_ENVIRONMENT="SqlServer"
Add-Migration AddChampActif -Context AppDbContext -OutputDir Migrations\SqlServer
```

► Générer une migration PostgreSQL

powershell

[Copier](#) [Modifier](#)

```
$env:DOTNET_ENVIRONMENT="PostgreSQL"
Add-Migration AddChampActif -Context AppDbContext -OutputDir Migrations\PostgreSQL
```

► Appliquer les migrations

powershell

[Copier](#) [Modifier](#)

```
$env:DOTNET_ENVIRONMENT="SqlServer"
Update-Database -Context AppDbContext
```

powershell

[Copier](#) [Modifier](#)

```
$env:DOTNET_ENVIRONMENT="PostgreSQL"
Update-Database -Context AppDbContext
```

Dans Program.cs on utilise le builder (chargement de l'appli, build, etc)

📁 Chargement automatique des fichiers `appsettings`

Quand tu fais :

csharp

```
var builder = WebApplication.CreateBuilder(args);
```

.NET Core charge automatiquement **dans cet ordre** :

1. `appsettings.json` (toujours)
2. `appsettings.{ENV}.json` selon la valeur de `DOTNET_ENVIRONMENT`
3. Toutes les variables d'environnement (pour override final)

```
$env:DOTNET_ENVIRONMENT = "SqlServer"
```

.NET va automatiquement charger :

- `appsettings.json`
- `appsettings.SqlServer.json`

Les clés présentes dans `SqlServer.json` **remplacent ou complètent** celles de base.

🔄 Résumé du cycle

Étape	Ce qu'il se passe
1. <code>\$env:DOTNET_ENVIRONMENT="PostgreSQL"</code>	Variable définie
2. Appel <code>Add-Migration</code> ou <code>dotnet run</code>	EF/.NET lit la variable
3. <code>Program.cs</code> lit <code>builder.Configuration</code>	Charge <code>appsettings.json</code> + <code>appsettings.PostgreSQL.json</code>
4. Tes conditions dynamiques s'exécutent	Ex: <code>UseNpgsql(...)</code> si <code>Provider == PostgreSQL</code>

Tuto 08/07 Les champs persos et Entity Personnel

Charger les données côté contrôleur

C#

```
1 public async Task<Personnel?> GetPersonnelAvecChampsPersosAsync(int id)
2 {
3     var personnel = await _context.Personnels.FirstOrDefaultAsync(p => p.Id == id);
4     if (personnel == null)
5         return null;
6
7     var typeInfos = await _context.TypeInformations
8         .Where(ti => ti.IdTable == "PERSONNEL")
9         .ToListAsync();
10
11     var infos = await _context.InformationGeneriques
12         .Where(i => i.IdAppartenance == id.ToString())
13         .ToListAsync();
14
15     var listeItems = await _context.TypeInformationListes
16         .Where(li => typeInfos.Select(ti =>
17             ti.NoTypeInformation).Contains(li.NoTypeInformation))
18         .GroupBy(li => li.NoTypeInformation)
19         .ToDictionaryAsync(g => g.Key, g => g.Select(x => x.Libelle).ToList());
20
21     personnel.ChampsPersos = typeInfos.Select(ti =>
22     {
23         var valeur = infos.FirstOrDefault(i => i.NoTypeInformation ==
24             ti.NoTypeInformation)?.Valeur ?? "";
25
26         return new CustomFieldItem
27         {
28             NoTypeInformation = ti.NoTypeInformation,
29             Libelle = ti.Libelle,
30             NoTypeDonnee = (int)ti.NoTypeDonnee,
31             Valeur = valeur,
32             ValeursPossibles = listeItems.ContainsKey(ti.NoTypeInformation) ?
33                 listeItems[ti.NoTypeInformation] : null
34         };
35     }).ToList();
36
37     return personnel;
38 }
```

Côté Vue Razor

C#

```
1 <h4>Champs personnalisés</h4>
2
3 @for (int i = 0; i < Model.ChampsPersos.Count; i++)
4 {
5     var champ = Model.ChampsPersos[i];
6     <div class="mb-3">
7         <label class="form-label">@champ.Libelle</label>
8
9         @if (champ.ValeursPossibles != null && champ.ValeursPossibles.Any())
10         {
11             <select class="form-select" name="ChampsPersos[@i].Valeur">
12                 @foreach (var val in champ.ValeursPossibles)
13                 {
14                     <option value="@val" selected="@((val == champ.Valeur))">@val</option>
15                 }
16             </select>
17         }
18         else
19         {
20             <input type="text" class="form-control" name="ChampsPersos[@i].Valeur">
```



```

        value="@champ.Valeur" />
21     }
22
23     <input type="hidden" name="ChampsPersos[@i].NoTypeInfoInformation"
        value="@champ.NoTypeInfoInformation" />
24     <input type="hidden" name="ChampsPersos[@i].Libelle" value="@champ.Libelle" />
25     <input type="hidden" name="ChampsPersos[@i].NoTypeDonnee" value="@champ.NoTypeDonnee" />
26 </div>
27 }

```

Enregistrer les données

C#

```

1  [HttpPost]
2  [ValidateAntiForgeryToken]
3  public async Task<IActionResult> Edit(int id, Personnel personnel)
4  {
5      if (id != personnel.Id)
6          return NotFound();
7
8      if (!ModelState.IsValid)
9          return View(personnel);
10
11     var existing = await _context.Personnels.FindAsync(id);
12     if (existing == null)
13         return NotFound();
14
15     existing.Nom = personnel.Nom;
16     existing.Prenom = personnel.Prenom;
17     existing.Email = personnel.Email;
18     existing.Actif = personnel.Actif;
19
20     foreach (var champ in personnel.ChampsPersos)
21     {
22         var existingChamp = await _context.InformationGeneriques
23             .FirstOrDefaultAsync(x =>
24                 x.IdAppartenance == id.ToString() &&
25                 x.NoTypeInfoInformation == champ.NoTypeInfoInformation);
26
27         if (existingChamp != null)
28         {
29             existingChamp.Valeur = champ.Valeur;
30         }
31         else
32         {
33             _context.InformationGeneriques.Add(new InformationGenerique
34             {
35                 IdAppartenance = id.ToString(),
36                 NoTypeInfoInformation = champ.NoTypeInfoInformation,
37                 Valeur = champ.Valeur
38             });
39         }
40     }
41
42     var aSupprimer = _context.InformationGeneriques
43         .Where(x => x.IdAppartenance == id.ToString() &&
44             !personnel.ChampsPersos.Select(c =>
45                 c.NoTypeInfoInformation).Contains(x.NoTypeInfoInformation));
46     _context.InformationGeneriques.RemoveRange(aSupprimer);
47     await _context.SaveChangesAsync();
48     TempData["Message"] = "Le personnel a bien été modifié.";
49     return RedirectToAction(nameof(Index));
50 }






```

Voyons ensuite à quoi ressemble actuellement Personnel sur [mon Visual Studio](#)

La version montrée est une classe si on adapte avec une base existante, autrement la plupart des attributs sont à enlever

C'est aussi pour commencer à se poser les questions de migration ancien Cosmos vers nouveau Cosmos

✓ Pourquoi garder les `[Column]` (et `[Table]`, `[Key]`, etc.) ?

Raisons	Explications
 Correspondance exacte	La base existante ne suit pas forcément les conventions EF Core (noms en PascalCase, <code>Id</code> , etc.). Les attributs garantissent que la classe <code>Personne1</code> correspond exactement aux noms de colonnes.
 Migrations stables	Sans attributs, EF pourrait mal interpréter les noms ou types, et générer de fausses différences dans les prochaines migrations.
 Interopérabilité multi-SGBD	Tu prévois un usage multiplateforme (SQL Server et PostgreSQL) → mieux vaut éviter les inférences hasardeuses.
 Prise en charge de <code>numeric(x, y)</code>	En particulier, <code>[Column(..., TypeName = "numeric(5,0)"]</code> permet de garder le type exact utilisé côté base. EF mapperait peut-être vers <code>int</code> ou <code>decimal</code> autrement.
 Lisibilité métier	Même si le code est un peu plus verbeux, il devient plus explicite sur le mapping avec la base réelle (utile pour ton équipe actuelle).