

CHAPITRE 3 :

Accès à une base de données avec Entity Framework Core

Plan du Chapitre

- Introduction
- Approches de développement avec EF Core
- EF Core Database Providers
- Installer EF Core
- Database First
- La Commande Scaffold-DbContext
- Code First
- La classe DbContext
- DbContextOptions
- Entity Framework Core DbSet
- Accès à une base de données SQL SERVER
- Repository Pattern
- Notion de Migration dans EF Core
- Conventions EF Core
- Mise à jour des données
- Validation du modèle
- File upload

Introduction

- Entity Framework Core est la nouvelle version d'Entity Framework après EF 6.x.
- Il s'agit d'une version open source, légère, extensible et multiplateforme de la technologie d'accès aux données Entity Framework.
- Entity Framework est un outil de mapping objet/relationnel (ORM).
- Il s'agit d'une amélioration d'ADO.NET qui offre aux développeurs un mécanisme automatisé pour accéder et stocker les données dans une base de données.
- EF Core est destiné à être utilisé avec les applications .NET Core. Cependant, il peut également être utilisé avec des applications basées sur le Framework .NET 4.5+.

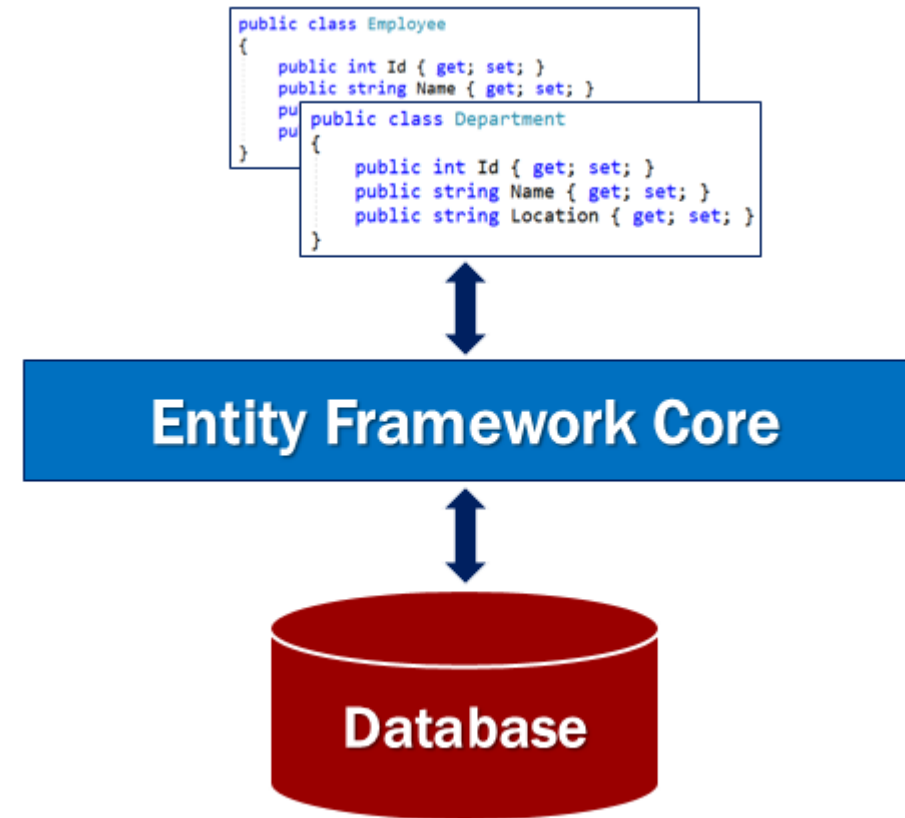
Introduction

- La figure suivante illustre les types d'applications pris en charge, les Framework .NET et les systèmes d'exploitation.

Application Types	<u>ASP.NET Core Applications</u> Web, API, Console, etc.	<u>.NET 4.5+ Applications</u> Console, WinForm, WPF, ASP.NET	Devices + IoT, Mobile, PC, Xbox, Surface Hub	<u>Mobile Application</u> Android, iOS, Windows
EF Core	EF Core	EF Core	EF Core	EF Core
Framework	.NET Core	.NET 4.5+	UWP	Xamarin
OS	Windows, Mac, Linux	Windows	Windows 10	Mobile

Approches de développement avec EF Core

- EF Core prend en charge deux approches de développement
 - Code-First
 - Database-First
- Si la base de données existe déjà, utiliser la technique Database First.
- EF Core cible principalement l'approche code First et fournit peu de support pour l'approche Database First.
- Le concepteur visuel ou l'assistant pour le modèle Database First présents dans EF6, ne sont pas pris en charge pour EF Core.



Approches de développement avec EF Core

- Dans l'approche code-first, EF Core API crée la base de données et les tables à l'aide de commande de migration en fonction des conventions et configurations fournies dans vos classes de domaine. Cette approche est utile dans la conception pilotée par domaine (DDD).



- Dans l'approche Database First, l'API EF Core crée les classes de domaine et de contexte en fonction du schéma de la base de données existante à l'aide de commandes EF Core.



EF Core Database Providers

- EF Core prend en charge de nombreuses bases de données relationnelles et même non relationnelles.
- EF Core peut le faire en utilisant des bibliothèques intégrés (plug-in libraries) appelées les fournisseurs de base de données (**Database Providers**).

Database	NuGet Package
SQL Server	Microsoft.EntityFrameworkCore.SqlServer
MySQL	MySql.Data.EntityFrameworkCore
PostgreSQL	Npgsql.EntityFrameworkCore.PostgreSQL
SQLite	Microsoft.EntityFrameworkCore.SQLite
SQL Compact	EntityFrameworkCore.SqlServerCompact40

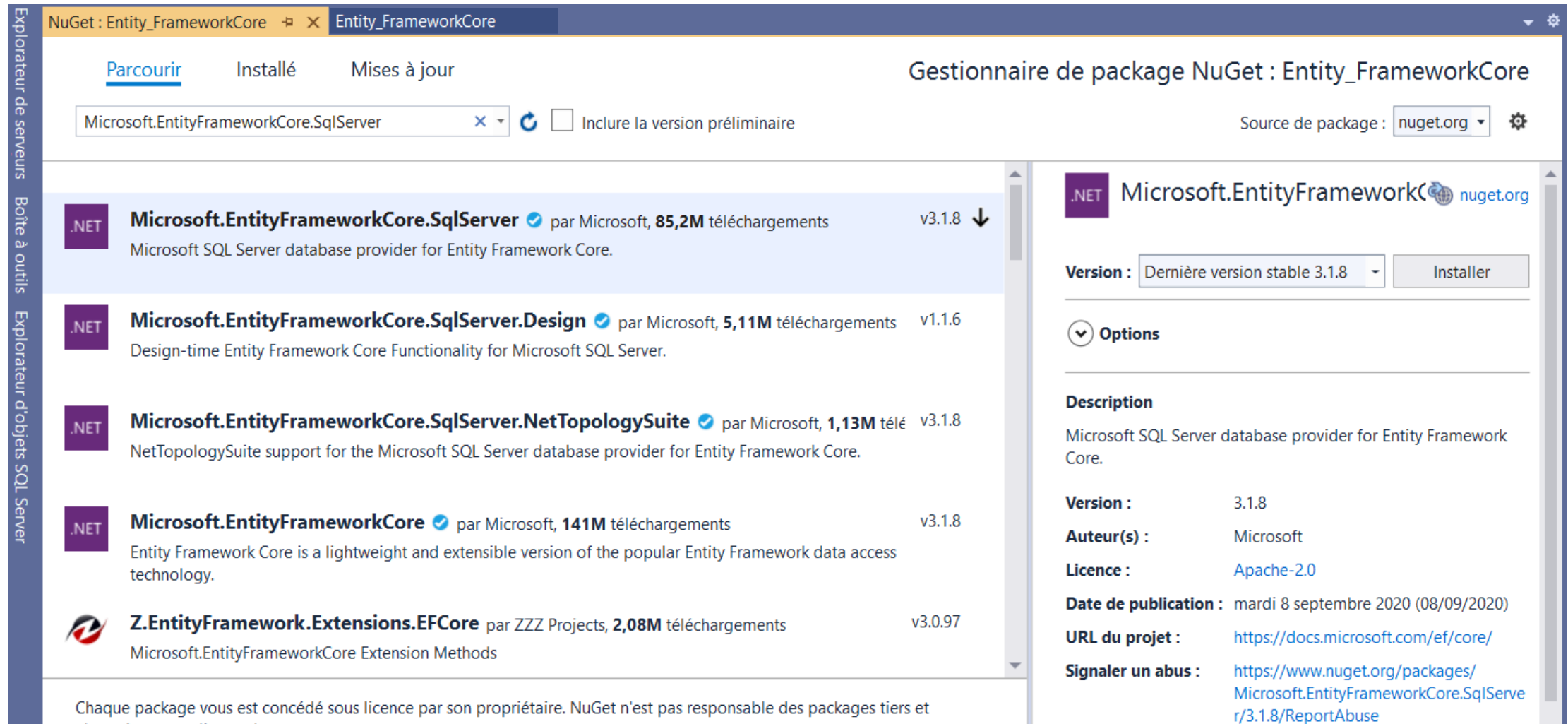


- Ces fournisseurs de bases de données sont disponibles sous forme de **packages NuGet** qu'il faut installer Dans votre projet.

Installer EF Core

- EF Core ne fait pas partie de .NET Core et du framework .NET standard.
- Il est disponible sous forme de package NuGet.
- Vous devez installer les packages NuGet suivantes pour utiliser EF Core dans votre application: **EF Core DB provider** et **EF Core tools**
- Sous Visual Studio, Click droit sur le nom du projet → Gérer les packages Nuget
- Pour une base de données SQL SERVER, Installer le package `Microsoft.EntityFrameworkCore.SqlServer`

Installer EF Core



NuGet : Entity_FrameworkCore Entity_FrameworkCore

Parcourir Installé Mises à jour

Gestionnaire de package NuGet : Entity_FrameworkCore

Microsoft.EntityFrameworkCore.SqlServer Inclure la version préliminaire

Source de package : nuget.org

Package	Version	Downloads
Microsoft.EntityFrameworkCore.SqlServer par Microsoft, 85,2M téléchargements	v3.1.8	↓
Microsoft.EntityFrameworkCore.SqlServer.Design par Microsoft, 5,11M téléchargements	v1.1.6	
Microsoft.EntityFrameworkCore.SqlServer.NetTopologySuite par Microsoft, 1,13M télé	v3.1.8	
Microsoft.EntityFrameworkCore par Microsoft, 141M téléchargements	v3.1.8	
Z.EntityFramework.Extensions.EFCore par ZZZ Projects, 2,08M téléchargements	v3.0.97	

Chaque package vous est concédé sous licence par son propriétaire. NuGet n'est pas responsable des packages tiers et

Microsoft.EntityFrameworkCore nuget.org

Version : Dernière version stable 3.1.8 Installer

Options

Description

Microsoft SQL Server database provider for Entity Framework Core.

Version : 3.1.8

Auteur(s) : Microsoft

Licence : Apache-2.0

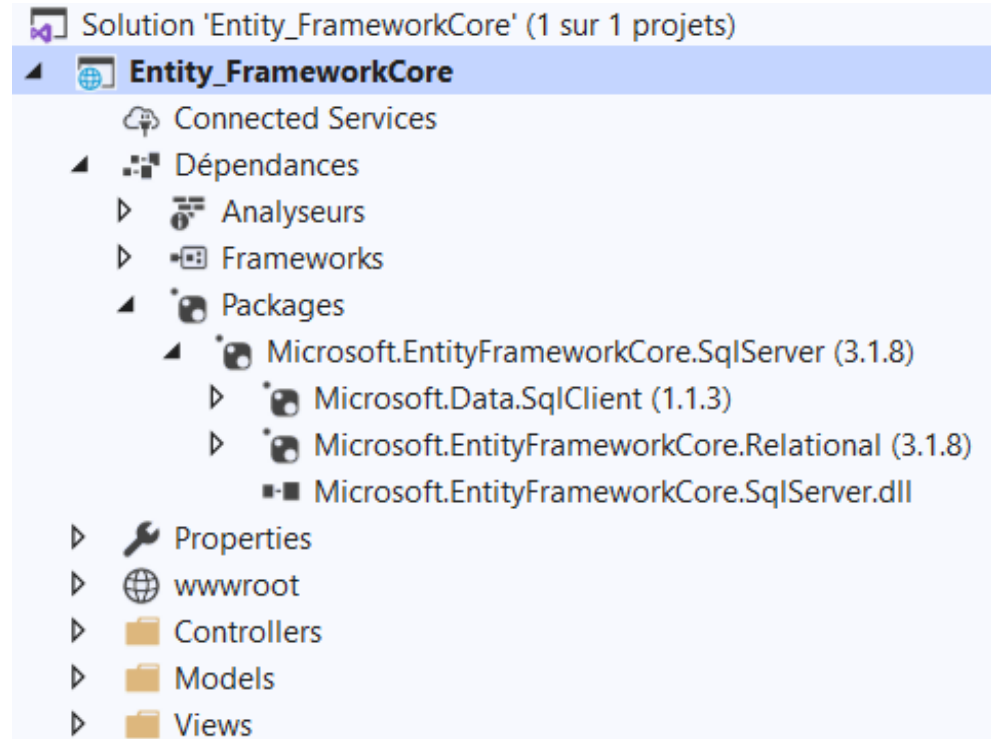
Date de publication : mardi 8 septembre 2020 (08/09/2020)

URL du projet : <https://docs.microsoft.com/ef/core/>

Signaler un abus : <https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.SqlServer/3.1.8/ReportAbuse>

Installer EF Core

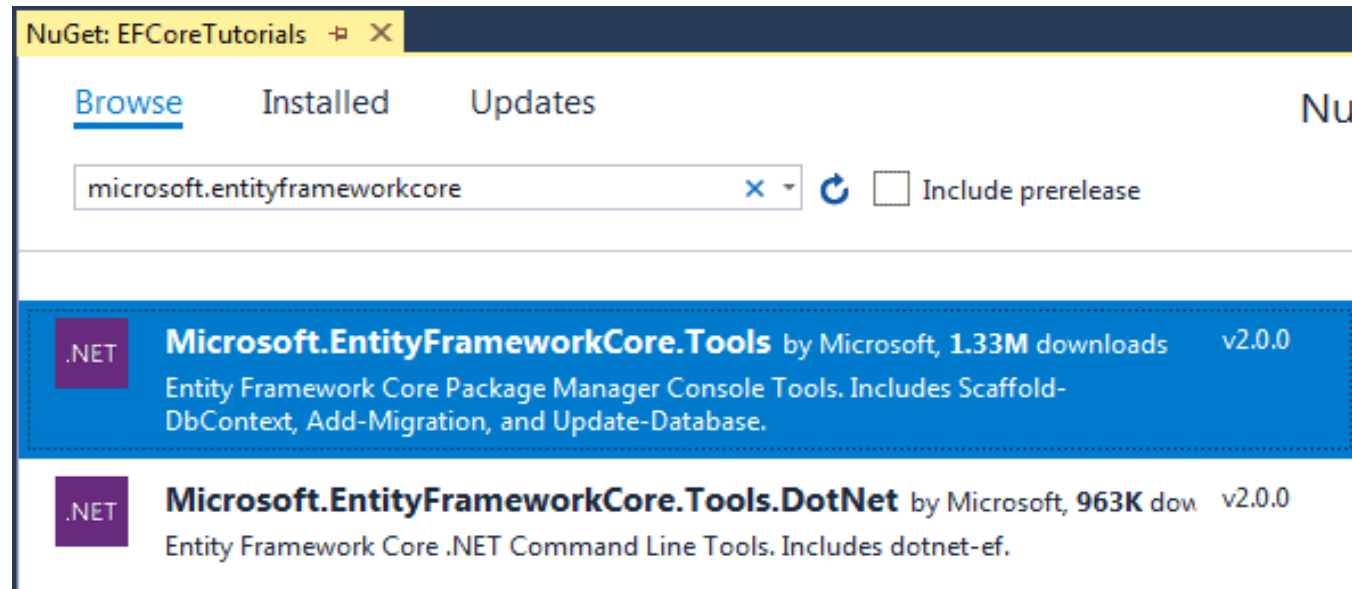
- Vérifier que les dépendances suivantes ont été bien installées :
 - Microsoft.EntityFrameworkCore.Relational
 - Microsoft.Data.SqlClient
- Vous pouvez aussi installer ce package via la console du gestionnaire de package (package manager console) via la commande suivante:



```
PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

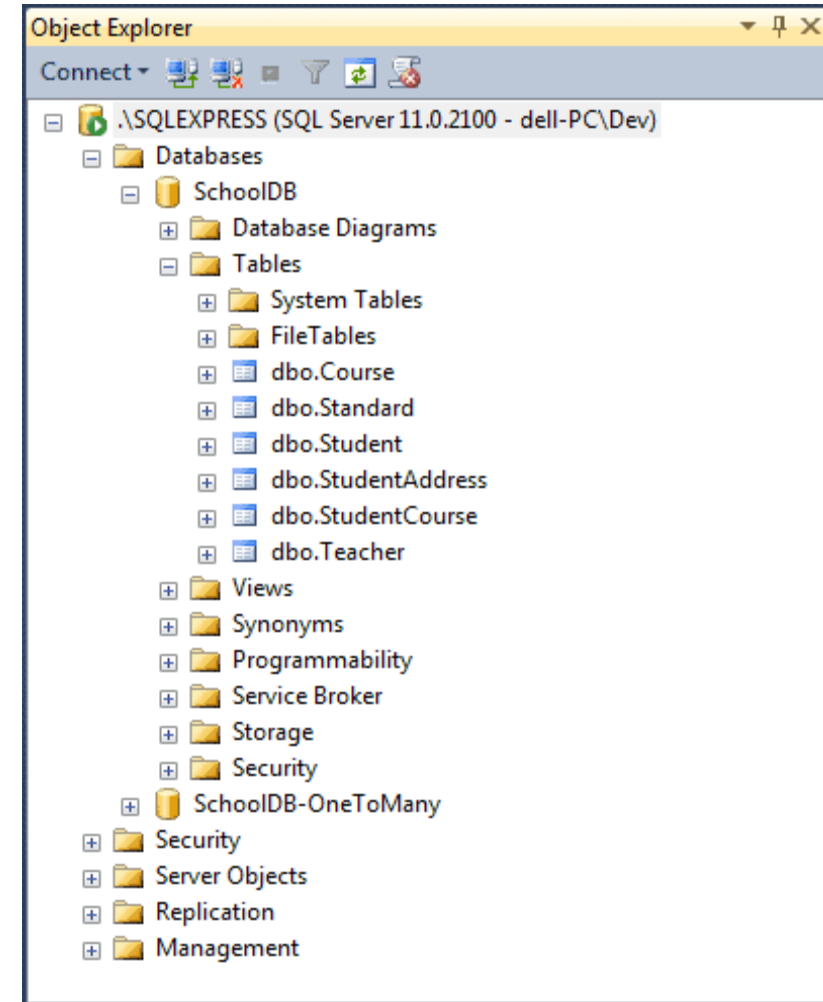
Installer EF Core Tools

- Pour exécuter les commandes EF Core à partir de la console du gestionnaire de package, recherchez le package Microsoft.EntityFrameworkCore.Tools à partir de NuGet et installez-le.
- Ce package permet d'exécuter les commandes EF de migration, de mise à jour de la base.



Database First

- EF Core ne supporte pas l'assistant de génération de modèle à partir de la base de données comme le cas dans EF6.
- EF Core utilise la technique de reverse engineering avec la commande **Scaffold-DbContext** pour générer les classes de modèle et la classe de contexte à partir du schéma de la base de données.
- Si on considère la base de données suivante sous SQL SERVER :



La Commande Scaffold-DbContext

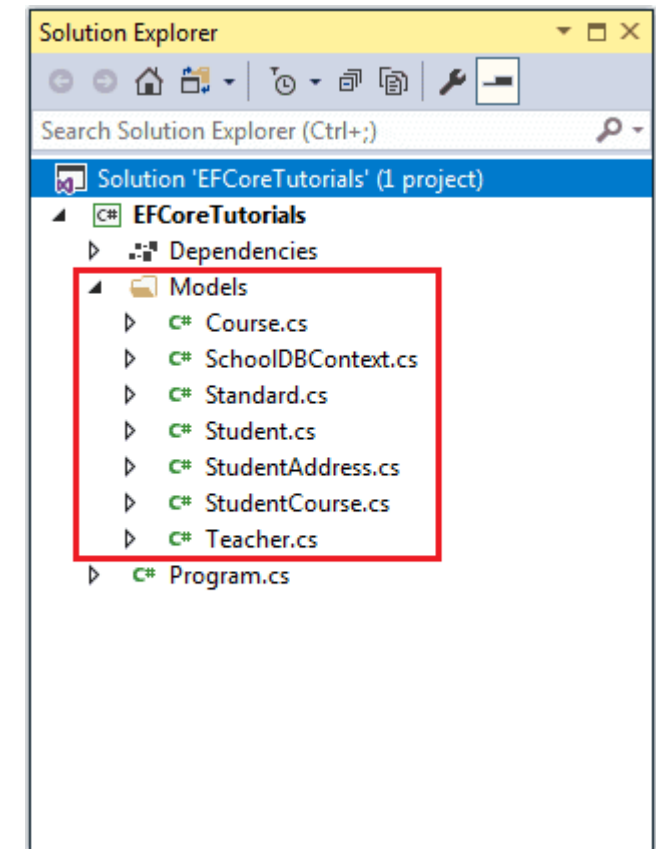
- Cette commande doit être exécutée dans **package manager console**

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

Exemple :

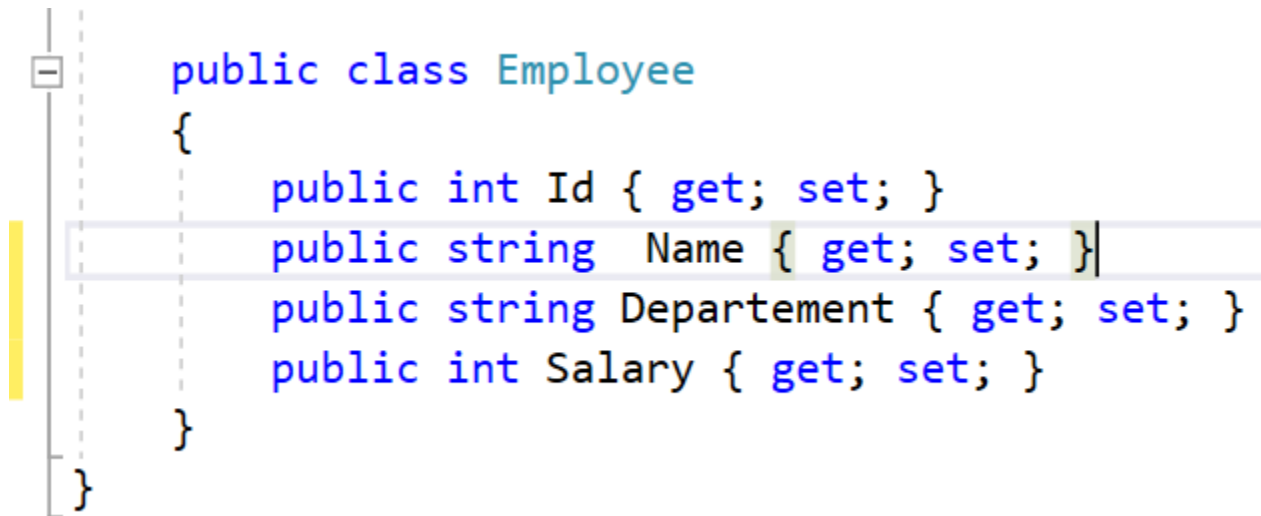
```
PM> Scaffold-DbContext "Server=.\SQLEXPRESS;Database=SchoolDB;
Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer
-OutputDir Models
```

- Une entité ou classe par table est générée, en plus la classe de Contexte SchoolDbContext est créée.



Code First

- Dans cette technique, on commence par créer nos classes de domaine.
- Ensuite, il faut créer la classe de contexte qui hérite de DbContext pour pouvoir communiquer avec une base de données.
- Créons cette classe Employee :



```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Departement { get; set; }
    public int Salary { get; set; }
}
```

The image shows a UML class diagram for the Employee class. The class is represented by a rectangle with a small square icon in the top-left corner. The class name 'Employee' is written in blue. The class contains four public attributes: 'Id' of type 'int', 'Name' of type 'string', 'Departement' of type 'string', and 'Salary' of type 'int'. Each attribute has a getter and a setter method, indicated by '{ get; set; }'. The class is enclosed in curly braces. A yellow vertical bar is visible on the left side of the diagram.

La classe DbContext

- Une des classes les plus importantes dans Entity Framework Core est la classe **DbContext**.
- C'est la classe que nous utilisons dans le code de l'application pour interagir avec la base de données sous-jacente.
- Gère la connexion avec la base de données et est utilisée pour récupérer et enregistrer des données dans la base de données.
- Pour utiliser la classe DbContext dans notre application nous créons une classe qui dérive de la classe **DbContext**.
- La classe **DbContext** se trouve dans l'espace de noms Microsoft.EntityFrameworkCore.

```
public class AppDbContext : DbContext
{
}
```

DbContextOptions

- Pour que la classe *DbContext* puisse effectuer son travail, elle a besoin d'une instance de la classe *DbContextOptions* .
- L' instance *DbContextOptions* transporte des informations de configuration telles que la chaîne de connexion, le fournisseur de base de données à utiliser, etc.
- Pour transmettre l' instance *DbContextOptions* , nous utilisons le constructeur comme indiqué dans l'exemple ci-dessous.

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
    {
    }
}
```


Entity Framework Core DbSet

- La classe *DbContext* inclut une *propriété DbSet<TEntity>* pour chaque entité du modèle.
- Dans cet exemple, la classe *AppDbContext* contient une seule *propriété DbSet<Employee>*.
- Nous utiliserons cette propriété *DbSet Employees* pour interroger et enregistrer des instances de la classe *Employee*.
- Les requêtes LINQ sur *DbSet <TEntity>* seront traduites en requêtes sur la base de données sous-jacente.
- Pour pouvoir se connecter à une base de données, nous avons besoin de la chaîne de connexion à la base de données.

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
    {
    }
    public DbSet<Employee> Employees { get; set; }
}
```

Accès à une base de données SQL SERVER

- Lorsque vous utilisez Entity Framework Core, l'une des choses importantes que nous devons configurer est le fournisseur de base de données
- Entity Framework Core prend en charge une grande variété de bases de données, y compris des bases de données non relationnelles.
- Le lien MSDN suivant contient la liste de toutes les bases de données prises en charge : <https://docs.microsoft.com/en-us/ef/core/providers/>
- Pour configurer et utiliser Microsoft SQL Server avec EF Core, il faut ajouter cette configuration dans la méthode ***ConfigureServices()*** du fichier ***Startup.cs***

Accès à une base de données SQL SERVER

```
public class Startup
{
    private IConfiguration _config;

    public Startup(IConfiguration config)
    {
        _config = config;
    }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContextPool<AppDbContext>(options =>
            options.UseSqlServer(_config.GetConnectionString("EmployeeDBConnection")));
    }

    // Reste du code
}
```

Accès à une base de données SQL SERVER

- Nous pouvons utiliser la *méthode* `AddDbContext()` ou `AddDbContextPool()` pour inscrire notre classe `DbContext` spécifique à l'application avec le système d'injection de dépendances ASP.NET Core.
- Avec la méthode `AddDbContextPool`, une instance du pool `DbContext` est fournie si disponible, plutôt que de créer une nouvelle instance.
- La méthode d'extension `UseSqlServer()` est utilisée pour configurer notre classe `DbContext` spécifique à l'application pour utiliser Microsoft SQL Server comme base de données.
- Pour nous connecter à une base de données SQL SERVER, nous avons besoin de la chaîne de connexion qui est fournie en paramètre à la méthode d'extension `UseSqlServer()`.

Accès à une base de données SQL SERVER

```
services.AddDbContextPool<AppDbContext>(options =>  
options.UseSqlServer(_config.GetConnectionString("EmployeeDBConnection")));
```

- La chaîne de connexion nommée "**EmployeeDBConnection**" doit être définie dans le fichier de configuration du projet **appsettings.json** plutôt que dans le code.
- Pour lire la chaîne de connexion à partir du fichier *appsettings.json*, nous utilisons la méthode **GetConnectionString()** du service **Iconfiguration**.
- Ajouter cette clé dans le fichier **appsettings.json** :

```
{  
  "ConnectionStrings": {  
    "EmployeeDBConnection": "server=(localdb)\\MSSQLLocalDB;database=EmployeeDB;  
Trusted_Connection=true"  
  }  
}
```

Repository Pattern

Repository Pattern in ASP.NET Core



```
public void ConfigureServices(IServiceCollection services)
{
    // Other Code...
    services.AddScoped<IEmployeeRepository, SQLEmployeeRepository>();
}
```

Repository Pattern

- Repository Pattern Interface

```
public interface IEmployeeRepository
{
    Employee GetEmployee(int Id);
    IEnumerable<Employee> GetAllEmployee();
    Employee Add(Employee employee);
    Employee Update(Employee employeeChanges);
    Employee Delete(int Id);
}
```

Repository Pattern

```
public class SQLEmployeeRepository : IEmployeeRepository
{
    private readonly AppDbContext context;

    public SQLEmployeeRepository(AppDbContext context)
    {
        this.context = context;
    }

    public Employee Add(Employee employee)
    {
        context.Employees.Add(employee);
        context.SaveChanges();
        return employee;
    }
}
```

```
public Employee Delete(int Id)
{
    Employee employee = context.Employees.Find(Id);
    if (employee != null)
    {
        context.Employees.Remove(employee);
        context.SaveChanges();
    }
    return employee;
}

public IEnumerable<Employee> GetAllEmployee()
{
    return context.Employees;
}
```


Repository Pattern

```
public Employee GetEmployee(int Id)
{
    return context.Employees.Find(Id);
}

public Employee Update(Employee employeeChanges)
{
    var employee =
context.Employees.Attach(employeeChanges);
    employee.State = EntityState.Modified;
    context.SaveChanges();
    return employeeChanges;
}
```

```
public class HomeController : Controller
{
    private IEmployeeRepository _employeeRepository;

    public HomeController(IEmployeeRepository employeeRepository)
    {
        _employeeRepository = employeeRepository;
    }

    public ActionResult Index()
    {
        var model = _employeeRepository.GetAllEmployee();
        return View(model);
    }
    // Rest of the code
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    // Reste du code
    services.AddScoped<IEmployeeRepository, SQLEmployeeRepository>();
}
```

Notion de Migration dans EF Core

- La migration est une fonctionnalité principale EF Core qui maintient le schéma de la base de données et nos classes de modèle de l'application synchronisées.



- Chaque fois que vous modifiez les classes de domaine, vous devez exécuter la migration pour maintenir le schéma de base de données à jour.
- Les migrations EF Core sont un ensemble de commandes que vous pouvez exécuter dans NuGet Package Manager Console ou dans dotnet Command Line Interface (CLI).

PMC Command	Usage
add-migration <migration name>	Creates a migration by adding a migration snapshot.
Remove-migration	Removes the last migration snapshot.
Update-database	Updates the database schema based on the last migration snapshot.
Script-migration	Generates a SQL script using all the migration snapshots.

Notion de Migration dans EF Core

- Vous pouvez utiliser la commande *get-help* avec l'une des commandes ci-dessus. Par exemple, *get-help Add-Migration* fournit une aide pour la commande Add-Migration.
- La commande suivante crée la migration initiale. InitialCreate est le nom de la migration.

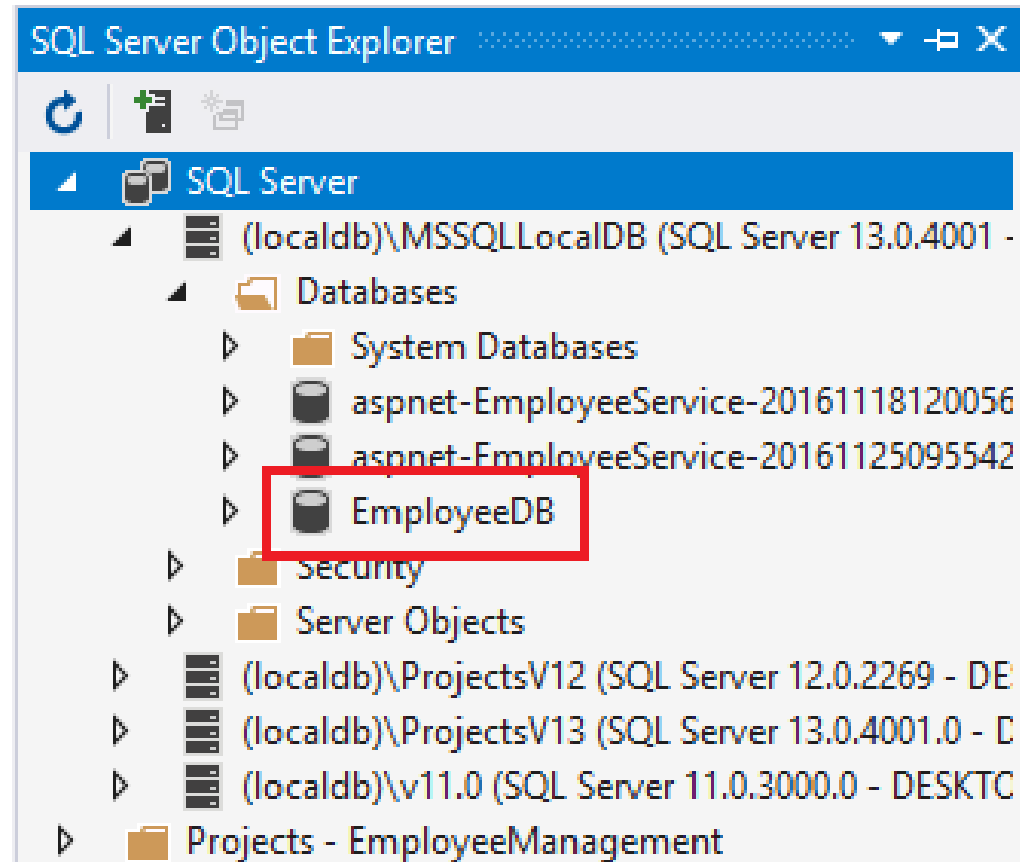
PM> Add-Migration InitialCreate

- Lorsque la commande ci-dessus se termine, vous verrez un fichier dans le dossier "Migrations" qui contient le nom InitialCreate.cs . Ce fichier contient le code requis pour créer les tables de la base de données.
- Pour mettre à jour la base de données, nous utilisons la commande *Update-Database*.

PM> Update-Database

Notion de Migration dans EF Core

- Allez à l'explorateur d'objets SQL SERVER, la base de données est créée avec la table Employee



Ajouter une Migration

- La classe Employee a la structure suivante :

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public string Department { get; set; }
}
```

- Si on veut changer la classe en ajoutant une propriété *PhotoPath*

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public string Department { get; set; }
    public string PhotoPath { get; set; }
}
```

Ajouter une Migration

- Pour grader la synchronisation entre la base de données et les classes de modèle, il faut lancer une Migration via la commande Add-Migration.

PM> Add-Migration AddPhotoPathToEmployees

- Fichiers du dossier de Migration créés :
 - [TimeStamp] _AddPhotoPathToEmployees.cs - Le nom de fichier est composé de TimeStamp, UnderScore et du nom de migration. Le nom de classe dans ce fichier a le même nom que le nom de migration. Cette classe contient 2 méthodes : **Up()** et **Down()**. La méthode **Up()** contient du code pour appliquer les modifications apportées à la classe de modèle au schéma de base de données sous-jacent. La méthode **Down()** contient du code pour annuler les modifications.
 - [DbContextClassName] ModelSnapshot.cs - Comme son nom l'indique, ce fichier contient l'état de votre modèle actuel. Ce fichier est créé lors de l'ajout de la première migration et mis à jour à chaque migration ultérieure.

Ajouter une Migration

- Pour appliquer la migration et mettre à jour la base de données, utilisez la commande **Update-Database**.
- Cette commande exécute le code dans la méthode Up() et applique les modifications aux objets de base de données sous-jacents.

PM> Update-Database

- La table __EFMigrationsHistory est créée dans la base de données, lorsque la première migration est exécutée.
- Cette table permet de suivre les migrations appliquées à la base de données.
- Il y aura une entrée pour chaque migration appliquée.

Supprimer une Migration

- Pour supprimer une migration qui n'est pas encore appliquée à la base de données, exécutez la commande `Remove-Migration`. Si toutes les migrations sont déjà appliquées, l'exécution de la commande `Remove-Migration` lève l'exception suivante:

The migration 'Latest_Migration_Name' has already been applied to the database. Revert it and try again.

- Comment supprimer une Migration appliquée à la base ?

Supposons qu'on a appliqué les 3 migration suivantes

- ✓ Migration_One
- ✓ Migration_Two
- ✓ Migration_Three

Et on veut annuler Migration_Two et Migration_Three et revenir à l'état de la base après Migration_one.

PM> Update-Database Migration_One // met la base à l'état après Migration_One

PM> Remove-Migration // permet de supprimer le fichier de code de la migration annulée

PM> Remove-Migration // et valide la suppression des deux dernières migrations

Configurer les classes de modèle

- Data Annotation Attributes:

```
[Table("StudentInfo")]
public class Student
{
    public Student() { }

    [Key]
    public int SID { get; set; }

    [Column("Name", TypeName="ntext")]
    [MaxLength(20)]
    public string StudentName { get; set; }

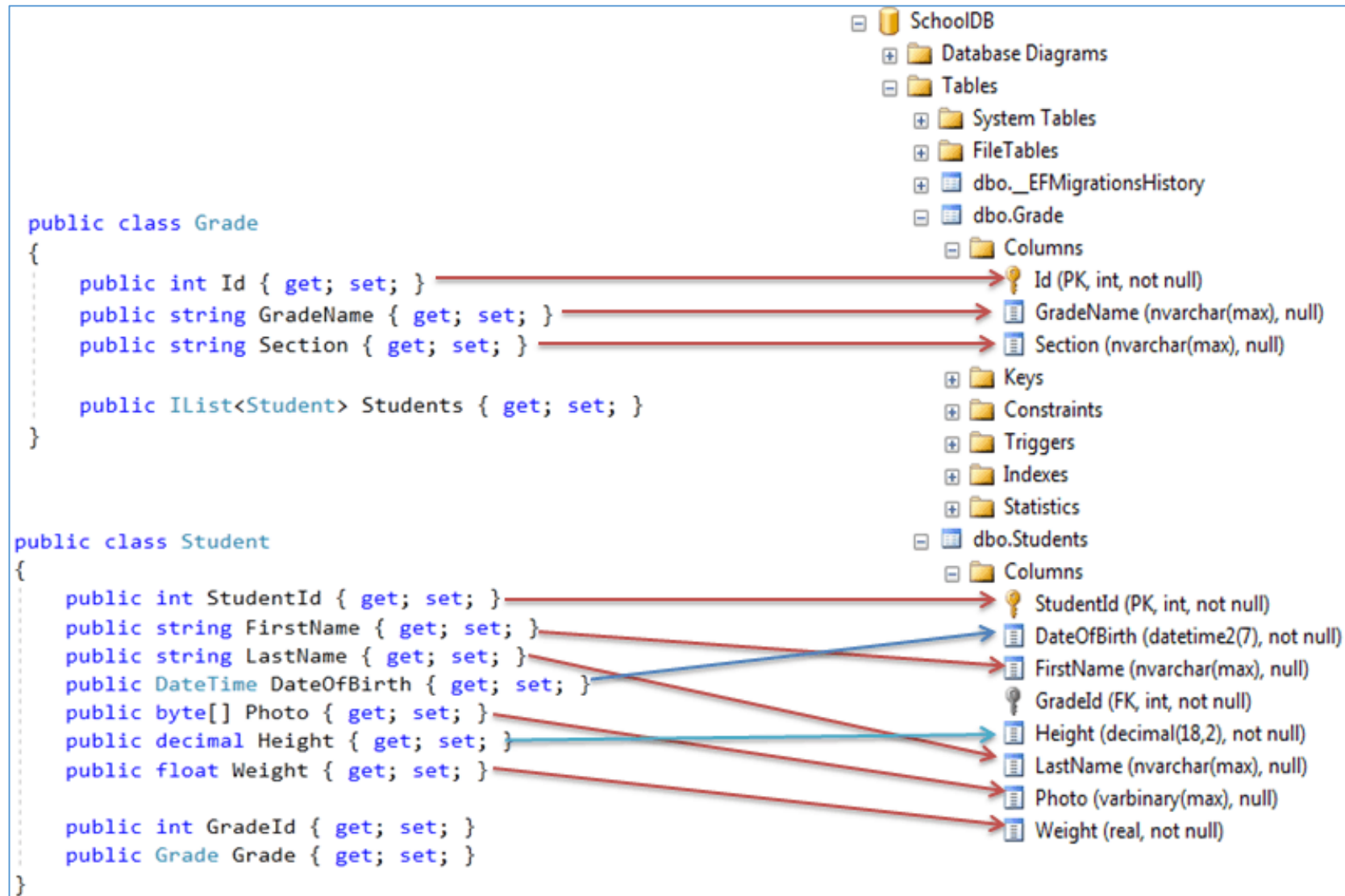
    [NotMapped]
    public int? Age { get; set; }

    public int StdId { get; set; }

    [ForeignKey("StdId")]
    public virtual Standard Standard { get; set; }
}
```

Conventions EF Core

- Conventions sur les colonnes:



Conventions EF Core

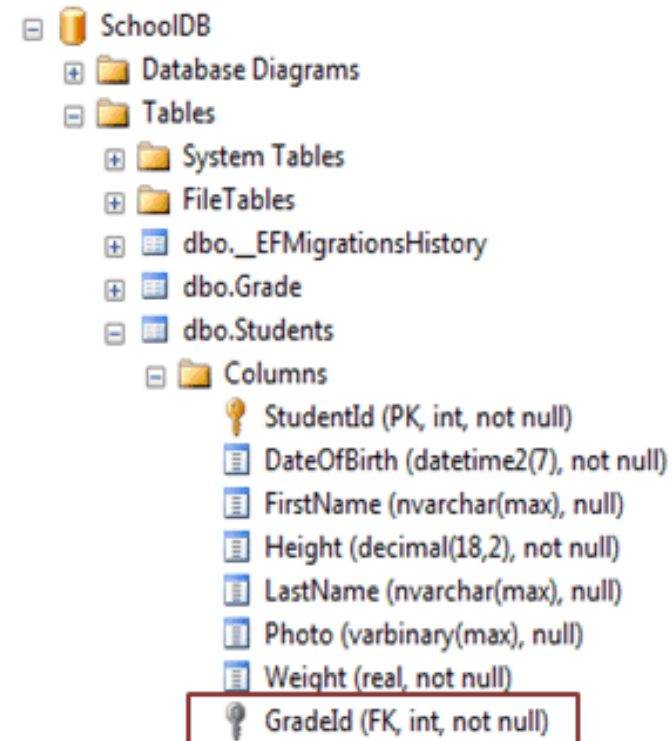
- Conventions sur les clés étrangères:

```
public class Student Dependent Entity
{
    public int StudentId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public int GradeId { get; set; } Foreign Key Property
    public Grade Grade { get; set; } Reference Property
}

public class Grade Principal Entity
{
    public int Id { get; set; } Primary Key Property
    public string GradeName { get; set; }
    public string Section { get; set; }

    public IList<Student> Students { get; set; }
}
```



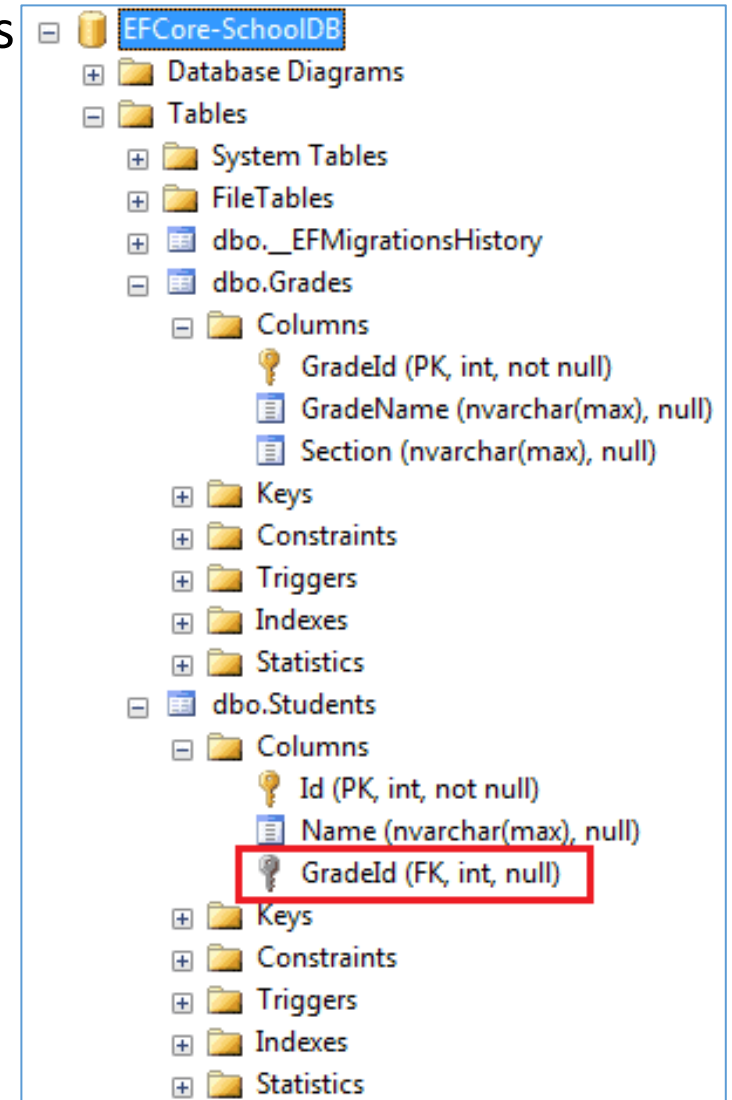
Conventions EF Core

- Conventions sur la relation one to many entre deux classes

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }
}
```



Conventions EF Core

- Autres Conventions : Relation one to many entre deux classes :

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeID { get; set; }
    public string GradeName { get; set; }

    public ICollection<Student> Students { get; set; }
}
```

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public int GradeId { get; set; }
    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }

    public ICollection<Student> Students { get; set; }
}
```

Conventions EF Core

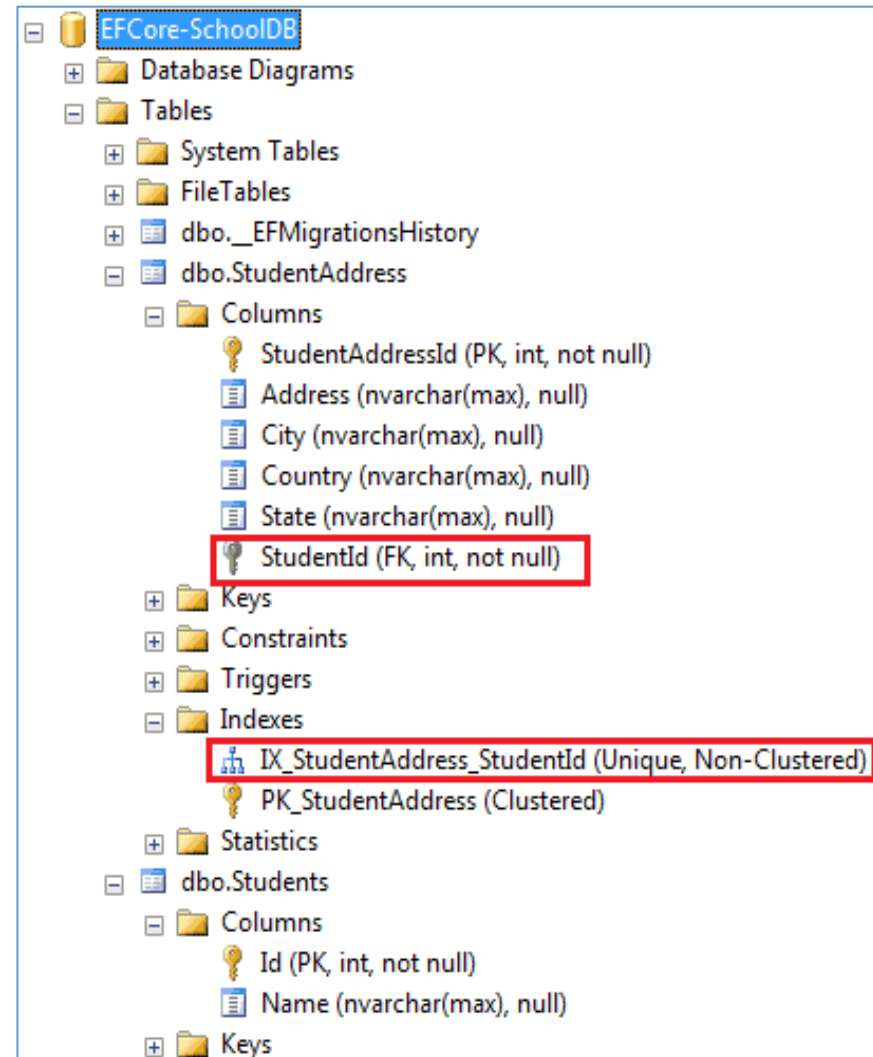
- Convention de la relation one to one entre deux classes :

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public StudentAddress Address { get; set; }
}

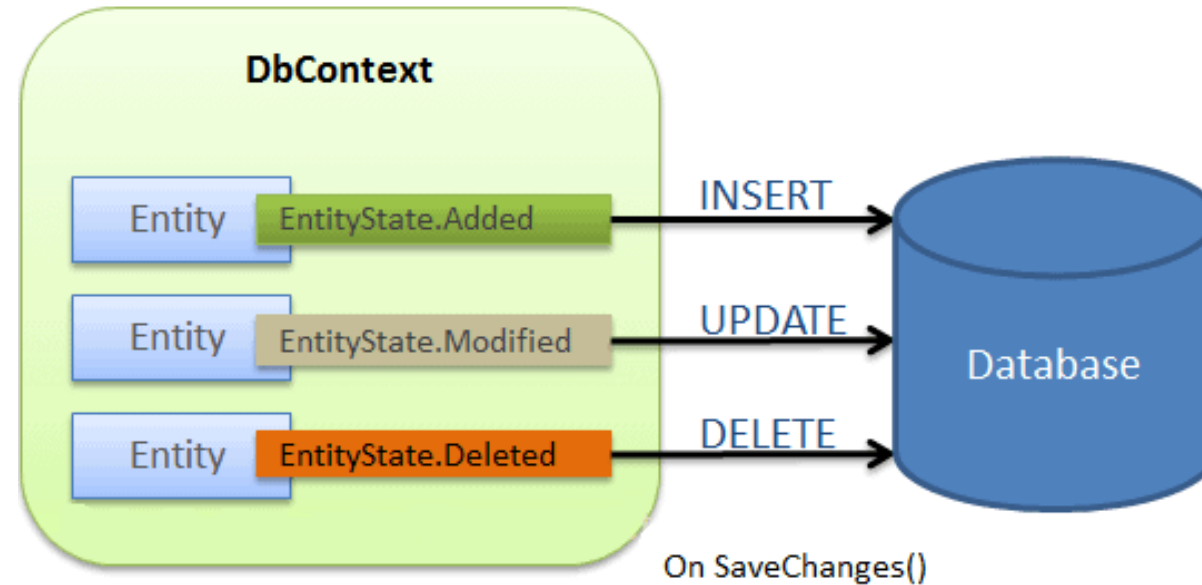
public class StudentAddress
{
    public int StudentAddressId { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

    public int StudentId { get; set; }
    public Student Student { get; set; }
}
```



Mise à jour des données

- En mode connecté:



- Insérer des données :

```
using (var context = new SchoolContext())
{
    var std = new Student()
    { FirstName = "Bill", LastName = "Gates" };
    context.Students.Add(std);
    // or // context.Add<Student>(std);
    context.SaveChanges(); }
}
```

Mise à jour des données

- Modification de données :

```
using (var context = new SchoolContext())
{
    var std = context.Students.First<Student>();
    std.FirstName = "Steve";
    context.SaveChanges();
}
```

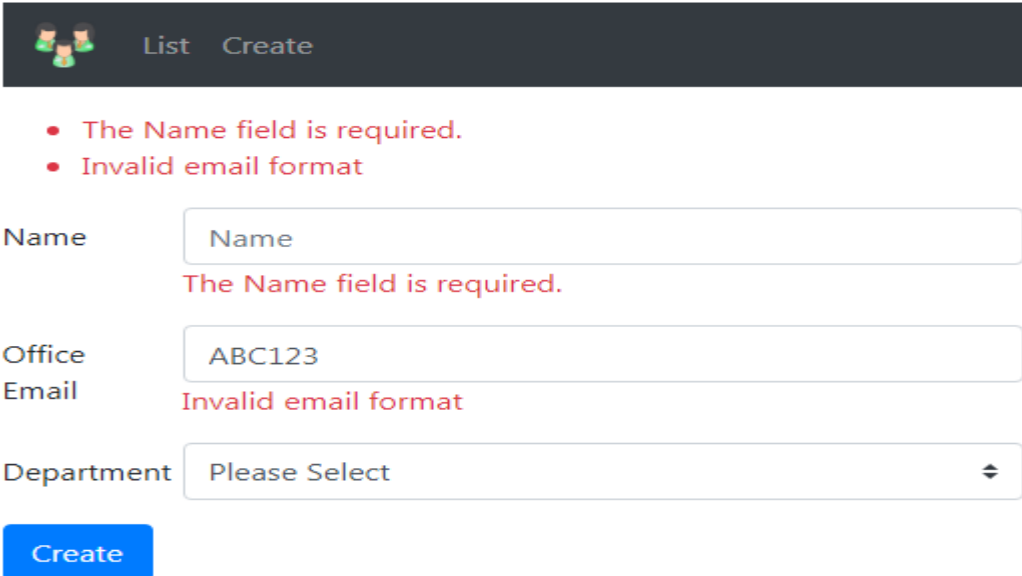
- Suppression de données :


```
using (var context = new SchoolContext())
{
    var std = context.Students.First<Student>();
    context.Students.Remove(std); // or //
    context.Remove<Student>(std);
    context.SaveChanges();
}
```


Validation de modèle dans ASP.NET CORE

- La validation des données permet de contrôler les valeurs saisies et d'afficher des messages d'erreur si les données ne sont pas conformes aux annotations indiquées.
- On considère le formulaire suivant, si on souhaite que la propriété Name soit obligatoire et l'email doit être valide, il faut ajouter les annotations de données à la classe Employé.
- Il faut ajouter le namespace *System.ComponentModel.DataAnnotations*

```
public class Employee
{
    public int Id { get; set; }
    [Required, MaxLength(50, ErrorMessage = "Name can not
exceed 50 characters")]
    public string Name { get; set; }
    [Display(Name = "Office Email")]
    [EmailAddress(ErrorMessage = "Invalid email format")]
    [Required]
    public string Email { get; set; }
    public string Department { get; set; }
}
```



 List Create

- The Name field is required.
- Invalid email format

Name
The Name field is required.

Office Email
Invalid email format

Department

Create

Validation de modèle dans ASP.NET CORE

- Dans le code de la méthode d'action Create, Utilisez la propriété ***ModelState.IsValid*** pour vérifier si la validation a échoué ou a réussi.
- Si la validation a échoué, nous renvoyons la même vue afin que l'utilisateur puisse fournir les données requises et soumettre à nouveau le formulaire.

```
[HttpPost]
public IActionResult Create(Employee employee)
{
    if (ModelState.IsValid)
    {
        Employee newEmployee = _employeeRepository.Add(employee);
        return RedirectToAction("details", new { id = newEmployee.Id });
    }

    return View();
}
```

Validation de modèle dans ASP.NET CORE

- Pour afficher les erreurs de validation, utiliser les TAG Helpers : ***asp-validation-for*** et ***asp-validation-summary***
- ***asp-validation-for*** affiche un message de validation pour une seule propriété de notre classe de modèle. Utiliser un élément **** pour afficher l'erreur.

```
<div class="form-group row">  
  <label asp-for="Name" class="col-sm-2 col-form-label"></label>  
  <div class="col-sm-10">  
    <input asp-for="Name" class="form-control" placeholder="Name">  
    <span asp-validation-for="Name"></span>  
  </div>  
</div>
```

- ***asp-validation-summary*** affiche un résumé de toutes les erreurs de validation du formulaire. Utiliser un élément **<div>** pour afficher la liste des erreurs. La valeur affectée peut être All, ModelOnly ou None.

```
<div asp-validation-summary="All">  
</div>
```

Validation de modèle dans ASP.NET CORE

```
<div asp-validation-summary="All">
</div>

<div>
  <label asp-for="Name"></label>
  <div>
    <input asp-for="Name">
    <span asp-validation-for="Name"></span>
  </div>
</div>

<div>
  <label asp-for="Email"></label>
  <div>
    <input asp-for="Email">
    <span asp-validation-for="Email"></span>
  </div>
</div>
```

Validation de modèle dans ASP.NET CORE

Attribut	Description
Required	Indique que la propriété est un champ obligatoire
StringLength	Définit une longueur maximale pour le champ de chaîne
Range	Définit une valeur maximale et minimale pour un champ numérique
RegularExpression	Spécifie que la valeur du champ doit correspondre à l'expression régulière spécifiée.
CreditCard	Spécifie que le champ spécifié est un numéro de carte de crédit.
CustomValidation	Méthode de validation personnalisée spécifiée pour valider le champ
EmailAddress	Valide avec le format d'adresse email
FileExtension	Valide avec extension de fichier
MaxLength	Spécifie la longueur maximale d'un champ de chaîne
MinLength	Spécifie la longueur minimale d'un champ de chaîne
Phone	Spécifie que le champ est un numéro de téléphone utilisant une expression régulière pour les numéros de téléphone.

File upload

- Pour comprendre la technique du file upload, on vous propose d'ajouter pour chaque employé son image qui sera stockée côté serveur dans le dossier wwwroot/images.
- Pour cela on ajoute dans la classe Employee une propriété nommée photopath

```
public class Employee
{
    public int Id { get; set; }
    [Required][MaxLength(50,ErrorMessage = "Taille Max 50 cc")]
    public string Name { get; set; }
    public string Departement { get; set; }
    [Range(300,5000,ErrorMessage = "Doit être entre 300 et 5000")]
    public int Salary { get; set; }
    public string PhotoPath { get; set; }
}
```

- Puis on procède à une Migration pour ajouter la nouvelle propriété dans la base.

File upload

- Pour pouvoir faire un upload il faut créer un attribut de type IFormFile.
- Il n'est pas pratique de déclarer cet attribut dans notre classe de modèle Employee. D'où la nécessité de créer une classe ViewModel.

```
public class CreateViewModel
{
    public string Name { get; set; }
    public string Department { get; set; }
    public int Salary { get; set; }
    public IFormFile Photo { get; set; }
}
```

- L'interface IFormFile est dans l'espace de noms Microsoft.AspNetCore.Http
- Le fichier à enregistrer sur le serveur est accessible via la liaison de modèle à l'aide de l'interface IFormFile .
- L'interface IFormFile contient les propriétés et méthodes suivantes

File upload

```
public interface IFormFile
{
    string ContentType { get; }
    string ContentDisposition { get; }
    IDictionary Headers { get; }
    long Length { get; }
    string Name { get; }
    string FileName { get; }
    Stream OpenReadStream();
    void CopyTo(Stream target);
    Task CopyToAsync(Stream target, CancellationToken cancellationToken = null);
}
```


File upload – Create Action Method

```
[HttpPost]
public ActionResult Create(CreateViewModel model)
{
    if (ModelState.IsValid)
    {
        string uniqueFileName = null;

        if (model.Photo != null)
        {
            string uploadsFolder =
                Path.Combine(hostingEnvironment.WebRootPath,
                    "images");

            uniqueFileName = Guid.NewGuid().ToString() + "_"
                + model.Photo.FileName;

            string filePath =
                Path.Combine(uploadsFolder, uniqueFileName);

            model.Photo.CopyTo(new FileStream(filePath,
                FileMode.Create));
        }
    }
}
```

```
Employee newEmployee = new Employee
{
    Name = model.Name,
    Salary = model.Salary,
    Departement = model.Department,
    PhotoPath = uniqueFileName
};

_employeeRepository.Add(newEmployee);

return RedirectToAction("details",
    new { id = newEmployee.Id });
}

return View();
}
```

File upload - Create View

```
@model Entity_FrameworkCore.ViewModels.CreateViewModel
@{
    ViewData["Title"] = "Create";
}
<h1>Create</h1>
<h4>Employee</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        @*To support file upload set the form element
        enctype="multipart/form-data" *@
        <form enctype="multipart/form-data" asp-action="Create">
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
        </form>
    </div>
</div>
```

```
</div>
<div class="form-group">
    <label asp-for="Department" class="control-label"></label>
    <select asp-for="Department" asp-items="@((new
    SelectList(ListeDep.Depts))" class="custom-select mr-sm-2" >
        <option value="">Selectionner une valeur</option> </select>
    <span asp-validation-for="Department" class="text-danger"></span>
</div>
<div class="form-group">
    <label asp-for="Salary" class="control-label"></label>
    <input asp-for="Salary" class="form-control" />
    <span asp-validation-for="Salary" class="text-danger"></span>
</div>
```

File upload - Create View

@* asp-for tag helper is set to "Photo" property. "Photo" property type is IFormFile so at runtime asp.net core generates file upload control (input type=file) *@

```
<div class="form-group row">
  <label asp-for="Photo" class="col-sm-2 col-form-label"></label>
  <div class="col-sm-10">
    <div class="custom-file">
      <input asp-for="Photo" class="form-control custom-file-input">
      <label class="custom-file-label">Choose File...</label>
    </div>
  </div>
</div>
<div class="form-group">
  <input type="submit" value="Create" class="btn btn-primary" />
</div>
</form>
</div>
</div>
```

```
</div>
```

```
<a asp-action="Index">Back to List</a>
```

```
</div>
```

@*This script is required to display the selected file in the file upload element*@

```
@section Scripts {
```

```
<script>
```

```
$(document).ready(function () {
    $('.custom-file-input').on("change", function () {
        var fileName = $(this).val().split("\\").pop();
        $(this).next('.custom-file-label').html(fileName);
    });
});
```

```
</script>
```

```
}
```