Entity Framework : créer la base de données à partir des objets Models.

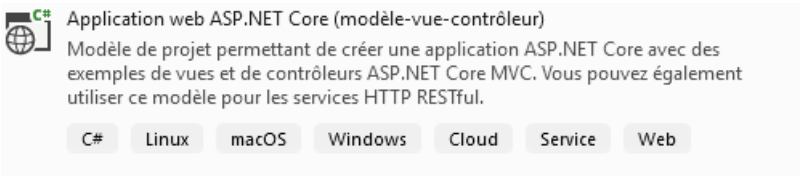
Créer un nouveau projet C#

La solution

Créer un dossier pour votre projet puis clic-droit Ouvrir avec Visual Studio

Fichier, Nouveau, Projet

Sélectionner le modèle



Donner un nom au projet, garder la version courante du framework et cliquer sur créer

Les dépendances nuget

Ajouter les dépendances au projet en cliquant sur Outils, Gestionnaire de package NuGet, Gérer les packages nugets pour la solution

```
MySql.EntityFrameworkCore
```

```
Microsoft.EntityFrameworkCore.Tools
```

```
Microsoft.EntityFrameworkCore
```

Vous pouvez voir les packages inclus en double cliquant sur le nom du projet (fichier nomProjet.csproj)

Le contexte

La classe contexte permet de créer la relation (session) avec la base de données. Elle étend la classe System.Data.Entity.DbContextDbContext. La classe de contexte est utilisée pour interroger ou enregistrer des données dans la base de données. Il est également utilisé pour configurer les classes de domaine, les mappages liés à la base de données, les paramètres de suivi des modifications, la mise en cache, les transactions, etc.

- Créer un dossier Data dans le projet
- Créer une classe nommée MyDBContext qui étend DbContext.
  ```
  public class MyDbContext:DbContext
  ```
- Insérer les dépendances
  Hint : pour insérer les dépendances, ctrl .
  ```
  using Microsoft.EntityFrameworkCore;
  ```
- Créer le construteur

Hint : pour créer un constructeur taper ctor et appuyer 2 fois sur la touche tabulation
Ce constructeur fera appel au constructeur hérité et lui passera les options de connexion à la base de données

```
public MyDbContext(DbContextOptions<MyDbContext> options):base(options)
 {
 }
```

- Ajouter le service DbContext au projet
  - Nous éviterons de mettre notre chaine de connexion directement dans la méthode de configuration de service. Nous allons créer une entrée dans le fichier json de configuration
  - L'entrée doit porter le nom `"ConnectionStrings"` obligatoirement avec un s. On peut configurer plusieurs bases de données pour différentes utilisations
  - Dans le fichier appsettings.json, ajouter l'entrée suivante. La base de données ne doit pas exister, elle va être créée lors de la migration

```
"ConnectionStrings": {
    "Default": "server=localhost;user=root;database=EFModelToBase;port=3306;ssl
mode=none" },
```

  - Dans le fichier Startup.cs, dans la méthode de configuration de services

Nous allons ajouter le service DbContext

```
public void ConfigureServices(IServiceCollection services)
{
            services.AddDbContext<MyDbContext>(options =>
options.UseMySQL(Configuration.GetConnectionString("Default")));

            services.AddControllersWithViews();
}
```

Les classes entité

- Dans le dossier Data, ajouter une classe
- Ajouter les propriétés
  Hint : pour ajouter des propriétés, taper prop et appuyer 2 fois sur tabulation

```
public class Personnes
{
    public int Id { get; set; }
    public string Prenom { get; set; }
    public string Nom { get; set; }
    public int Age { get; set; }
}
```

- Ajouter la classe entité au contexte

Dans le fichier MyDbContext.cs, ajouter la propriété correspondante à notre entité.

```
public DbSet<Personnes> Personnes { get; set; }
```

La migration

Dans la console du gestionnaire de packages

- Créer la migration :

```
Add-migration NomDeLaMigration
```

Le nom de la migration doit commencer par une majuscule car il sera le nom de la classe de la migration

- Un dossier Migration a été créé avec une classe HorodatageNomDeLaMigration.cs

Cette classe contient toutes les informations permettant la création de la base de données.

```
public partial class Initiale : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Personnes",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("MySQL:ValueGenerationStrategy",
MySQLValueGenerationStrategy.IdentityColumn),
                    Prenom = table.Column<string>(type: "text", nullable: true),
                    Nom = table.Column<string>(type: "text", nullable: true),
                    Age = table.Column<int>(type: "int", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Personnes", x => x.Id);
                });
        }

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

Les types générés ne correspondent pas forcément à nos habitudes de créations comme par exemple l'utilisation de type text pour le nom au lieu de varchar

- Modifier les types générés dans le contexte
    - Soit dans la classe MyDbContext.cs, en surchargeant la méthode OnModelCreating pour changer le type de l'âge en short au lieu de int

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<Personnes>(e => e.Property(o =>
o.Age).HasColumnType("tinyint(1)").HasConversion<short>());
        }
```

    - Soit dans la classe entité, en ajoutant une annotation(décoration) sur la propriété

```
public class Personnes
{
    public int Id { get; set; }

    [MaxLength(50)]
    public string Prenom  { get; set; }

    [MaxLength(50)]
    public string Nom { get; set; }

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

- Supprimer la migration

```
remove-migration
```

- Recréer la migration

```
add-migration Initiale
```

Voici le résultat obtenu

```
    public partial class Initiale : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Personnes",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("MySQL:ValueGenerationStrategy",
MySQLValueGenerationStrategy.IdentityColumn),
                    Prenom = table.Column<string>(type: "varchar(50)", maxLength: 50,
nullable: true),
                    Nom = table.Column<string>(type: "varchar(50)", maxLength: 50,
nullable: true),
                    Age = table.Column<short>(type: "tinyint(1)", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Personnes", x => x.Id);
                });
        }

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

La création de la base et de la table

Dans la console du gestionnaire de packages, appliquer la migration à la base de données

```
update-database
```

On peut voir qu'une table supplémentaire a été créée, elle contient les différentes migrations appliquées à cette base.