Base de données avec Entity Framework Core

- New Page

New Page

Interaction avec la base de donnée

L'interaction avec la base de donnée se fait via l'ORM (Object-Relationnal Mapper) officiel de Microsoft, Entity Framework Core.

Installer Entity Framework Core

Il faut installer les packages nugets via les commandes suivantes :

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 6.0.0
dotnet add package Pomelo.EntityFrameworkCore.MySql --version 6.0.0
```

Il faut également installer l'outil en ligne de commande d'EF Core pour gérer les migrations :

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

Le `DbContext`

Il faut ensuite créer notre classe `DbContext`, point d'entrée de notre base de donnée. Créez un dossier `Database` et dans ce dossier une classe `ApplicationDbContext` qui étend `DbContext`:

```
using Microsoft.EntityFrameworkCore;

namespace net_web_tuto.Database {

    public class ApplicationDbContext : DbContext {

        public ApplicationDbContext(DbContextOptions options) : base(options) {
        }
    }

}
```

Il faut ensuite rattacher notre `ApplicationDbContext` à notre application, dans votre `Program.cs` (après la ligne `var builder = WebApplication.CreateBuilder(args);`), rajoutez :

```
// Database
builder.Services.AddDbContext<ApplicationDbContext>(options => {
    string databaseHost = Environment.GetEnvironmentVariable("DATABASE_HOST");
    string databaseName = Environment.GetEnvironmentVariable("DATABASE_NAME");
    string databaseUsername = Environment.GetEnvironmentVariable("DATABASE_USERNAME");
    string databasePassword = Environment.GetEnvironmentVariable("DATABASE_PASSWORD");

    var connectionString =
$"server={databaseHost};database={databaseName};user={databaseUsername};password={databasePassw
ord}";
    options.UseMySql(connectionString, ServerVersion.AutoDetect(connectionString));
});
```

On récupère les identifiant depuis des variables d'environnement, c'est plus projet que des les avoir dans le code. Vous devriez aussi rajouter des `using` au début du fichier :

```
using Microsoft.EntityFrameworkCore;
using net_web_tuto.Database;
```

Définir les entités

Revenons à notre `ApplicationDbContext` auquel on va rajouter la méthode suivante pour définir nos entités :

```
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {

        }
```

Il faut aussi rajouter en attributs un `DbSet` qui va resprésenter notre table :

```
public DbSet<Person> Persons {get;set;}
```

Ensuite dans la méthode `OnModelCreating`, on peut définir notre entité (on a ajouté un champs `Id` de type `Guid` à la classe `Person` pour servir d'identifiant dans la base :

```
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
```

```
    modelBuilder.Entity<Person>().HasKey(person => person.Id);
        modelBuilder.Entity<Person>().Property(person => person.FirstName);
        modelBuilder.Entity<Person>().Property(person => person.LastName);
    }
```

On peut également définir des contraintes sur les attributs de l'entité, avec des méthodes comme `HasMaxLength`, `HasMinLength` chainées après l'appel à `Property`. Vous pouvez trouver plus de précisions sur la documention d'EF Core

Migrations

Après avoir défini notre modèle, il faut créer notre migration vers la base de donnée et l'exécuter. On va utiliser l'invite de commande EF Core pour ça.

Dans un premier temps on définit notre variables d'environnement (exemple avec Powershell, la syntaxe change selon votre shell):

```
$Env:DATABASE_HOST = "localhost"
$Env:DATABASE_NAME = "nomDeLaDb"
$Env:DATABASE_USERNAME = "userDeLaDb"
$Env:DATABASE_PASSWORD = "mdpDeLaDb"
```

> Avant de faire une migration, faite bien attention d'avoir éteint votre application, sinon vous allez avoir des err

Ensuite, exécutez la commande suivante pour créer la migrations :

```
dotnet ef migrations add maMigration
```

Enfin pour exécuter la migration sur votre base :

```
dotnet ef database update
```

Si vous vous connectez à votre base avec un client, vous deviez voir le schéma déployé.

Utiliser le `DbContext` dans le controleur

>

> Attention, si vous utiliser Rider, reportez les varibales d'environnement dans le fichier `Properties/launchSettings.json` dans l'objet `environmentVariables` de l'objet profile que vous utilisez pour lancer votre app (si vous avez un doute vous pouvez le mettre dans les deux).

Pour interagir avec la base de donnée depuis notre controleur, on peut demander au framework de nous l'injecter :

```
using Microsoft.AspNetCore.Mvc;
using net_web_tuto.Database;
using net_web_tuto.Models;

[Route("persons")]
public class PersonsController : Controller {

    private readonly ApplicationDbContext context;

    public PersonsController(ApplicationDbContext context){
        this.context = context;
    }

}
```

Créons maintenant un endpoint servant une vue avec :

- Une liste des personnes dans la base
- Un formulaire pour ajouter une personne

```
@foreach (Person person in @Model)
{
    <p>@person.FirstName - @person.LastName</p>
}

<form action="/persons" method="post">
    <input name="FirstName" type="text"/>
    <input name="LastName" type="text"/>
    <input value="Create" type="submit"/>
</form>
```

D'abord, pour récupérer les personnes, nous allons accèder à notre `DbSet` et le transformer en liste, pour EF Core, cela revient à faire une requête pour récuprer tous les enregistrement de la

table :

```
[HttpGet]
public IActionResult PersonsPage(){
    List<Person> persons = this.context.Persons.ToList();

    return View("persons", persons);
}
```

Ensuite, on peut ajouter notre endpoint d'insertion :

```
[HttpPost]
public IActionResult AddPerson(Person p){
    this.context.Persons.Add(p);
    this.context.SaveChanges();

    return new RedirectResult("/persons");
}
```

On ajoute la `Person` récupérée depuis le formulaire au `DbSet`, puis on appelle `SaveChanges` afin de valider la transaction sur la base.

Enfin, on retourne une redirection vers notre endpoint d'affichage pour réafficher la page avec les données mises à jours.

> Vous pouvez regarder dans la console les requêtes effectuées par EF Core sur la base de données, qui correspondent à nos appels

On va ensuite ajouter une vue pour voir une seule personne, avec un lien dans la liste. Créons d'abord notre endpoint :

```
[HttpGet("{id}")]
public IActionResult GetOnePerson(Guid id){
    Person person = this.context.Persons
        .First(p => p.Id == id);

    return View("person", person);
}
```

On utilise un paramètre de route pour récupérer l'identifiant. Ensuite, la méthode `First` prend un prédicat (fonction retournant un booléen) sur les personnes et permet de filtrer la liste pour ne

retourner que le premier enregistrement qui valide le prédicat. EF Core compile cette méthode sous la forme d'une clause `WHERE` dans la requête SQL. L'appel à `First` permet ensuite de récupérer le premier élément du résultat (ici logiquement il n'y en a qu'un).

Pour faire un filtrage sur la table et récupérer une collection d'enregistrement, il faut utiliser la méthode `Where` qui prend également un prédicat, mais retourne tous les enregistrement qui le valident :

```
        List<Person> persons = this.context.Persons
            .Where(p => p.FirstName.Contains("e"))
            .ToList();
```

On retourne ensuite la vue suivante :

```
@Model.FirstName - @Model.LastName
```

On rajoute dans notre vue `persons` des liens vers notre nouveau endpoint :

```
@foreach (Person person in @Model)
{
    <p><a href="/persons/@person.Id">@person.FirstName - @person.LastName</a></p>
}
```

Les relations avec EF Core

Pour créer des relations, il faut utiliser les méthodes `HasOne`, `HasMany`, `WithOne`, `WithMany` dans la méthode `OnModelCreating` du `DbContext`. Les méthodes "Has" permettent de définir le premier dans de la relation, et les méthodes With, permettent de définir l'autre coté. Cela permet de créer toutes les relations possibles. Exemple avec une relation "1 lié à n" (avec un livre lié à des pages) :

```
        modelBuilder.Entity<Book>()
            .HasMany(book => book.Pages) // Premier sens
            .WithMany(page => page.Book); // Sens inverse
```

On est pas obligé de passer de paramètre dans la méthode de sens inverse, si on a pas besoin de la navigation en sens inverse (ici : si on a pas besoin d'avoir de référence au livre dans la page).

Ensuite, pour récupérer dans une requête le conteu de la propriété de navigation, il faut utiliser `Include` :

```
context.Books
    .Include(book => book.Pages)
```

```
    .ToList();
```