Formation Entity Framework Core

Antoine DIEUDONNE

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Sommaire

- Qu’est ce que EF Core ?
- Les différents packages Nuget
- Code First vs Database First
- l’annotation de nos modèles
- Le contexte de données
- Sécuriser notre chaine de connexion
- Notre première migration
- Le Database First
- Le CRUD de Base
  - INSERT
  - SELECT
  - UPDATE
  - DELETE

Sommaire

- Les relations de cardinalité
- Les Propriétés de navigation
- Les différentes cardinalités
  - Le One-to-One
  - Le One-to-Many
  - Le Many-to-Many
- Insérer des données avec liaisons
- Récupérer des données avec liaisons
- Le Repository Pattern
  - BaseRepository
  - IRepository<T>
  - Les différents Repositories

Qu’est ce que Entity Framework Core ?

- Entity Framework Core est le framework leader d’accès au données porté par Microsoft dans le contexte d’une application réalisée avec .NET

- Son développement date de **2008** avec la première version d’Entity Framework. Suite à cela, le framework a beaucoup évolué pour donner en **2016** Entity Framework Core, permettant une utilisation cross-platform. Depuis **2021**, Entity Framework 6 est disponible

- EF Core est un **Objet Relational Mapper** (ORM). Grâce à lui, il est plus aisé de travailler avec les données. Son utilisation est donc principalement dûe à un désir d’améliorer la productivité et de nous connecter à un ou plusieurs types de bases de données rapidement et efficacement.

- Son utilisation en entreprise n’est plus à prouver. Des **millions d’applications professionnelles** ont vues le jour et se servent quotidiennement d’EF Core pour fonctionner. Ses performances lui permettant de gérer des quantités de données conséquentes avec de grands gains en matière de performances.

Les différents packages Nuget

- **EntityFrameworkCore.SqlServer** / **EntityFrameworkCore.Sqlite** : Pour permettre l'accès aux données dans notre application et la liaison au bon type de base de données

- **EntityFrameworkCore.Tools** : Pour permettre la création de migrations et la modification de la base de données dans la console de gestionnaire de packages

- **EntityFrameworkCore.Design** : Pour permettre la visualisation des données ainsi que la création potentielle de composants de façon automatisée

Code First vs Database First

- EF Core peut s'utiliser de deux façons majeures :

- Il est possible de créer nos tables à la volées en fonction de nos besoins de développement (en se basant sur les classes que l'on a besoin d'utiliser dans notre application). Cette approche est dite **Code First**

- A contrario, il est possible de nous servir d'une base de données préexistante et de nous en servir pour construire la base de notre application (nos classes) qui seront ainsi générées automatiquement par les outils d'Entity Framework pour nous. Cette approche est dite **Database First**

L’annotation de nos modèles

- Dans le cas où nous souhaitons réalisé une implémentation d’EF Core de façon Code First, il va donc nous falloir débuter par la création de nos modèles.

- Ces modèles (les classes) vont devoir être **annotés** afin d’optimiser au maximum leur passage dans la base de données (par exemple, le nombre de caractères que prendront les textes sous la forme de tables)

- Certaines annotations sont optionnelles, mais par convention ajoutées. Par exemple, l’annotation **[Key]** est optionnelle si l’on respecte les convention de nommage de nos Ids. Les conventions sont régies par l’utilisation d’un nom de propriété tel que **Id** ou **NomClasseId.**

- Pour les clés étrangères, il nous suffit d’avoir une agrégation d’un Id nommé **ClasseBId** et d’une propriété de type **ClasseB** qui se voit **virtualisée** (par soucis de surcharge de la propriété par EF Core en aval). Encore une fois, le non respect des conventions force à utiliser l’annotation **[ForeignKey(« nomForeignKey »)]**

Le contexte de données

- Une fois nos modèles configurés, il nous faut utiliser une classe qui nous servira de base à la liaison des données. Cette classe portera le nom **d'ApplicationDbContext**, encore une fois dans un soucis de convention. Elle devra **hériter de DbContext** (qui est disponible de par l'utilisation du package NuGet **EntityFrameworkCore**)

- Le contexte de donnée pourra posséder un ou plusieurs **DbSet<T>** qui seront des propriétés publiques permettant l'accès à nos futures tables (chaque DbSet doit rassembler une série de classes pour pouvoir les manipuler indépendamment les unes des autres)

- Il nous faudra ensuite ajouter une ou méthode surchargée => **OnConfiguring :** Cette méthode est utilisée pour modifier les options de base de notre contexte de données. Par exemple, ici, il nous est possible d'ajouter notre chaine de connexion et d'informer le contexte de données que nous utilisons une connexion de type SQLServer (via la méthode **.UseSqlServer()**)

Où mettre notre chaine de connexion ?

- Il est également possible de sécuriser un peu plus notre application en plaçant notre chaine de connexion dans un fichier appellé **secrets.json** Pour ce faire, il nous faudra avoir recourt au package **Microsoft.Extensions.Configuration.UserSecrets** Ce package nous offre la possibilité de stocker dans un fichier au format JSON contenu à un emplacement généré par Visual Studio sur notre machine.

- Le fichier ne sera pas exporté lors d'un push sur Git, mais il sera lu dans le cas où l'on le demande lors de l'initialisation du programme.

- Pour ce faire, il suffit d'ajouter quelques lignes de code dans notre méthode **OnConfiguring()**

```
var config = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddUserSecrets<Program>()
    .Build();

optionsBuilder.UseSqlServer(config.GetConnectionString("SQLServer"));
```

Première migration

- Entity Framework fonctionne sur le principe des migrations. Une migration est un fichier de configuration permettant à un ORM de connaître les modifications à apporter dans une base de données pour assurer le lien entre le code et les données de notre programme.
- Par convention, notre première migration devra se nommer Initial et ne contenir que les modèles de données correctement configurés au départ. Grâce à cela, il est possible lors de modifications futures de repartir toujours de ce point initial lors de changements éventuels de nos données
- Les migrations serviront en cas de modifications. Il est possible ainsi d'avoir plusieurs branches de modifications de la structure de notre base de données via l'ajout de multiples migrations. Il suffit ensuite de spécifier à l'ORM lesquelles prendre en compte et lesquelles ignorer lors de la mise à jour de la base de données.

- Pour ajouter une migration, il suffit de faire la commande **Add-Migration <nomDeLaMigration>** dans la console de gestionnaire de package. La suppression d'une migration crée par erreur se fera via la commande **Remove-Migration**
- Une fois nos scripts de migrations réalisés, il suffit d'utiliser la commande **Update-Database** pour mettre à jour la base de données à la dernière migration (Si l'on veut spécifier à quelle migration on veut se positionner, il est possible d'ajouter le nom du script en paramètre de la commande => **Update-Database <IDScript>**)

Le Database First

- Dans le cas où nous aurions déjà une base de données à laquelle nous souhaitons nous brancher dans la réalisation de notre programme, il est tout à fait possible de nous épargner une grande partie du travail et de créer à la volée notre contexte de données ainsi que nos modèles via le **scaffolding**.

- Pour ce faire, il va nous falloir utiliser dans la console de gestionnaire de packages la commande **Scaffold-DbContext –provider <nom.package.nuget.provider> -connection <chaine de connexion>**

Insérer des données

- Pour l'**ajout de données** dans notre base de données, il suffit, avec EF Core, d'utiliser la méthode **.Add()** dans la propriété de type **DbSet<T>** de notre contexte de données. Une version asynchrone => **.AddAsync()** est également disponible.

- Une fois fait, il nous faudra **sauvegarder les changements**, soit de façon synchrone avec la méthode **.SaveChanges()**, soit de façon asynchrone via la méthode **.SaveChangesAsync()**, qu'il faudra bien entendu attendre via l'utilisation d'un **await**.

Sélectionner des données

- Dans le cadre de la **visualisation** des données, les méthodes qui vont nous intéresser cette fois-ci se nomment :
- **.ToList()** et **.ToListAsync()**, qui permettent d'obtenir l'ensemble des données de notre **DbSet<T>** sous la forme d'une liste facilement exploitable par notre programme. Il est possible de précéder cette méthode d'une méthode **.Where()** dans laquelle on aurait placé un **prédicat** dans le but de filtrer l'ensemble des données.

- **.FirstOfDefault()** et **.FirstOfDefaultAsync()** serviront quant à elles à la récupération d'une seule et unique donnée via le passage en paramètre d'un **prédicat** sous la forme d'une expression lambda). Si aucun élément ne correspond, on aura la valeur par défaut de notre type rechercher (**null** pour un objet donc).
- **.SingleOrDefault()** et **.SingleORDefaultAsync()** vont tenter de trouver l'unique valeur dans notre base de données et lever une exception en cas de la présence de multiples valeurs similaires.
- **.Find()** et **.FindAsync()** vont renvoyer la valeur correspondant à la clé passée en paramètre de la méthode.

Modifier des données

- Si l'on souhaite maintenant recourir à une **édition** des données, il va nous falloir utiliser d'autres méthodes offertes par EF Core. Via l'utilisation du tracking géré par le framework, il nous suffit de modifier les classes puis d'avoir recourt aux méthodes :

- **.SaveChanges()** ou **.SaveChangesAsync()** pour sauvegarder toutes les modifications ayant eu lieu sur l'ensemble de notre contexte de données

- Il existe également les méthodes **.Update()** et **.UpdateAsync()** pour sauvegarder les changement indépendamment des éléments les uns par rapport aux autres si par exemple nous n'avons pas recourt au tracking automatisé.

Supprimer des données

- Enfin, si l'on veut cette fois-ci réaliser une **suppression** des données, il va nous falloir utiliser la méthode **.Remove()** sur le **DbSet<T>** comme on le ferait avec une liste.

- Ensuite, il nous faudra avoir recourt à la méthode **.SaveChanges()** ou **.SaveChangesAsync()** pour voir s'opérer la suppression dans la base de données.

Liste de contacts – 60 min

EXERCICE

- En vous servant de Entity Framework Core et de SQLServer

- Réaliser une application permettant à un utilisateur de pouvoir manipuler une liste de contacts. Ces contacts seront composés des propriétés suivantes : leur **nom**, leur **prénom**, leur **date de naissance**, leur **âge**, leur **genre**, leur **numéro de téléphone** et leur **adresse mail**.

- L'utilisateur de programme (qui bénéficiera d'une IHM pour l'utilisation en mode console) pourra avoir accès à la liste des contacts et ainsi pouvoir voir, modifier, supprimer ou ajouter un ou des contacts à cette liste.

- Ces changements devront être effectifs en base de données à chaque modification, de façon à avoir un mode « connecte » entre notre programme et notre base de données.

Les relations de cardinalités

- Lorsque l'on manipule des données, il est évident que l'on va à un moment ou un autre, rencontrer des relations de cardinalités. Par cardinalité, nous entendons les relations de type :

- **One-to-One** : Une donnée est liée à une autre donnée contenue dans une table voisine
- **One-to-Many** : Une données est liées à plusieurs données contenues dans une table voisine
- **Many-to-One** : Plusieurs données de notre tables peuvent être liées à une donnée dans une table voisine
- **Many-to-Many** : Plusieurs données de notre tables peuvent être liées à plusieurs données dans une table voisine

Les propriétés de navigation

- Pour Entity Framework Core, les relations de cardinalités vont se traduire par l'utilisation de **propriétés de navigation** dans nos modèles de données.

- Ces propriétés de navigations seront adjointes d'une clé étrangère, qui est optionnelle mais vivement recommandée. Pour créer une propriété de navigation, il suffit de créer une propriété du type de la classe avec laquelle nous souhaitons nous lier. Ces propriétés seront précédées du mot-clé **virtual** pour faciliter la surcharge des setters / getters par EF Core.

- La clé étrangère devra pour faciliter le travail d'EF Core se nomme selon la convention **NomPropNavId** sous peine de devoir être précédée de l'annotation **[ForeignKey(« NomPropNav »)]** pour fonctionner comme telle.

One-to-One

- Pour les relations de type **One-to-One**, il n’y a rien de plus simple. Il suffit de créer une propriété de navigation dans le modèle qui se voit être lié à un autre. Par exemple, un chien peut avoir comme propriété de navigation un maître pour que chaque chien appartienne à un maître.

One-to-Many

- Lors d'une relation de type **One-to-Many**, il nous faudra d'un côté avoir une propriété de navigation du type de l'élément lié à plusieurs éléments (un chien aura une propriété de type maître).

- De l'autre côté, il nous faudra une propriété de navigation d'un type conteneur (par exemple **ICollection<T>**) pour permettre par exemple à notre maître l'accès à l'ensemble de ses chiens.

Many-to-Many

- Lors d'une relation de type **Many-to-Many**, il nous faudra cette fois-ci passer par la création d'une nouvelle classe, qui servira à la liaison. Par exemple, chaque vente peut concerner des produits et des acheteurs, en plus de posséder une date, un moyen de paiement, etc.

- Pour ce faire, nous aurons deux propriétés de navigations ainsi que deux clés étrangères. Dans notre exemple, nous aurions donc deux propriétés (une de type **Produit**, et une du type **Acheteur**) adjointes de leurs clés étrangères (une **ProduitId** et une **AcheteurId** donc)

Insérer les données liées

- Lorsque l'on veut du coup ajouter les données liées à d'autres données, il va nous suffire d'utiliser soit les propriétés de navigations, soit les clés étrangères qui seront alimentées dans la donnée de base (par exemple notre maître pourra voir sa **liste de chiens** remplie ou les chiens voir leur propriété **Maître** ou **MaîtreID** alimentée).

- Une fois fait, il ne nous reste plus qu'à avoir recourt à la bonne vieille méthode .**Add**() ou .**AddAsync**() sur notre élément puis à sauvegarder les changements avec .**SaveChanges**() ou .**SaveChangesAsync**() pour qu'EF Core se charge du reste.

Récupérer les données liées

- Lorsque l'on veut afficher les données à l'utilisateur, il est fréquent que l'on ait besoin de faire appel aux données reliées aux premières. Pour ce faire, EF Core va encore une fois nous offrir des méthodes pour réaliser ce genre de scénarios :

- Pour récupérer en même temps que la données les liaisons, il suffit de précéder les méthodes de type .**ToList**() de .**Include**() dans laquelle nous mettrons en paramètre une lambda reliée à la propriété de navigation.

- Si l'on veut ensuite avoir les données liées à celle que l'on vient de lier, il faudra faire suite notre première méthode de .**ThenInclude**() avec en paramètre une nouvelle lambda, encore une fois avant la méthode de type .**ToList**() ou .**FirstOrDefault**()

Video Store – 60 min

EXERCICE

- En vous servant de Entity Framework Core et de SQLServer

- Réaliser un code permettant à un utilisateur d'accéder à une application permettant de gérer des films ainsi que des clients tout en permettant à ces dernier d'emprunter les films.

- Les clients auront comme informations leur **nom**, **prénom**, **numéro de téléphone** et leur **adresse mail**. Les films posséderont comme infos leur **titre**, leur **réalisateur**, une **description**, un **score** (sur 5) et un **prix** à la location.

- Un client doit pouvoir emprunter plusieurs films à la fois, et chaque emprunt devra se voir attribuer une **date** d'emprunt qui sera alors commune à tous les films empruntés.

- En **bonus**, vous pourrez faire en sorte que le **programme vérifie que le client ne possède pas déjà des emprunts en retard (14 jours ou plus)** sous peine de ne pas l'autoriser à réemprunter des films avant d'avoir rendu ceux qu'il avait préalablement empruntés.

Le Repository Pattern

- Jusqu'à maintenant, nous nous sommes contentés d'utilisé notre contexte de données tel quel et de faire appel aux méthodes .**Include**() ainsi que .**ThenInclude**() à de potentiels multiples endroit.

- En plus d'être la porte ouverte à la faute typographique, cette méthodologie n'est pas très prisée en entreprise car elle évite la répartition des tâches tout en réduisant la maintenabilité et la croissance de notre application. Pour y remédier, nous allons donc avoir recourt à un architecture pattern : le **Repository Pattern**

- Ce schéma de construction architectural se base sur l'utilisation d'interface de type **IRepository<T>**, d'une classe abstraite **BaseRepository** ainsi que de multiples classes de type **Repository** pour séparer l'accès à nos différentes tables dans des classes dédiés, permettant ainsi de ne pas permettre l'accès direct à la classe gérant les habitants à la gestion des immeubles par exemple.

BaseRepository

- Le rôle de la classe abstraite **BaseRepository** est simplement de contenir la variable **ApplicationDbContext** encapsulée de façon protégée, pour ensuite permettre à tous les futurs repositories d'en posséder l'accès par héritage.

IRepository<T>

- L'interface générique **IRepository<T>** va de son côté contenir l'ensemble des méthodes d'accès aux données qu'il va nous falloir implémenter. Il en existe trois variantes majeures :

- La **première** ne contient que des méthodes de CRUD réalisées de façon **synchrone**, elle est la plus simple à utiliser

- La **seconde** se sert des mêmes fonctionnalités mais les réalise de façon **asynchrone**, ce qui permet d'éviter les problèmes de latence dans le cas de réalisation d'applications plus poussées

- Enfin, la **troisième** est, en plus de l'utilisation des méthodes de CRUD asynchrone, capable d'être utilisée pour la **programmation évènementielle** via la présence d'évènements auxquels on peut se brancher et qui seront déclenchés par exemple lors de l'ajout, de la modification, etc… des éléments en base de données.

Les classes de Repository

- Chaque modèle de donnée devra donc posséder sa propre classe de Repository, qui se nommera par convention **NomModèleRepository** Cette classe va hériter à la fois de la classe abstraite **BaseRepository** et de l'interface **IRepository<T>** en en spécifiant le type générique pour y placer le type du modèle dont on souhaite réaliser le Repository.

- Une fois cela fait, il nous suffira d'implémenter les méthodes permettant le CRUD, de sorte à ne plus avoir à l'avenir à réécrire les différentes méthodes .**Include**() et .**ThenInclude**() mais de simplement faire appel au Repository et à ses méthodes telles que **.GetAll()** ou **.Delete()**

eLibrary – 120 min

EXERCICE

- En vous servant de Entity Framework Core, du Repository Pattern et de SQLServer

- Réaliser une application permettant à un utilisateur d'accéder à bibliothèque numérique.

- Cette bibliothèque devra permettre la consultation d'une liste d'ouvrages qui possèderont un **titre**, un **descriptif**, un **ISBN**, un **nombre de page**, une **catégorie**, un **auteur** et une **maison d'édition**.
- L'auteur possèdera comme propriété un **nom**, un **prénom**, une **date de naissance** ainsi que le **nombre de ses ouvrages**
- Une maison d'édition possèdera un **nom** ainsi qu'un **nombre d'ouvrages édités**.

- L'utilisateur de l'application devra pouvoir réaliser un **CRUD sur les trois types de données** contenues dans l'application (chaque donnée devra posséder son propre Repository) et se voir offrir un menu de navigation entre les différentes sections de l'application. La suppression d'un auteur ou d'une maison d'édition devra supprimer tous les ouvrages qui pourraient leur être liés.

Forum de Nouvelles – 180 min

TP

- En vous servant de Entity Framework Core, du Repository Pattern et de SQLServer

- Réaliser une application permettant à un utilisateur de naviguer dans un forum de nouvelles. Le forum contiendra des **questions**, des **réponses**, des **commentaires** et des **utilisateurs.**

- Les questions auront comme sous-ensembles des **réponses**, un **titre**, une **description**, un **score** et un **posteur**.
- Les réponses auront comme propriétés des **commentaires**, un **score**, une **description**, un **posteur** et un **titre**
- Les commentaires auront un **titre**, un **posteur**, une **description** et un **score**.
- L'utilisateur sera composé d'un **nom**, d'un **prénom**, d'un **téléphone** et d'une **adresse mail**.

- Le **score des questions, des réponses et des commentaires changera idéalement en fonction de leur manipulation** (par exemple, à chaque consultation d'une question, son score s'incrémentera, de même qu'à chaque modification)