Notes sur le module : *LInQ et Entity.*

Armel Pitelet

1er février 2022

Table des matières

Dans le folder Cours : ce qui a été fait et projeté, Exercice : ce que j'ai pris au passsage.

1 LInQ

Mot clef : **LInQ**.

Language intergrated Query : Façon d'interroger une collection comme

- Set (liste qui garantie l'unicité les valeurs) : fonctionne via itérateur et pas indexeur. Très performant pour des petites collections, moins pour les grosses car il faut parcourir toute la collection.
- ICollection ou IList : la liste classique (ICollection est plus haut dans la chaine d'interface). IList : list avec accesseur numérique (indexeur) et des méthodes supplémentaire par rapport à ICollection.
- IDictionnary : système clef-valeur. Clef unique. Plus rapide que l'accès via une liste (ex d'un dictionnary de int, string) mais plus complexe à gérer. On s'en sert plutôt quand la clef est complexe (string ou objet). Utilisé avec des clefs int si la gestion de l'int n'est pas accessible (et donc pas connu).

Voir aussi pour la liste des méthodes LinQ : https://docs.microsoft.com/fr-fr/dotnet/api/system.linq.enumerable?view=net-6.0

Remarque : *Sur les types.*

Il faut connaitre la correspondance de type en c# et SQL : UserFakeRepositoryTest. Le type doit être choisi en fonction de l aplace que l'on veut allouer en mémoire (en octets).

Remarque : *sur text et ntext.*

Il vaut mieux éviter de s'en servir car ils ne sont pas limité en taille.

Pour la suite on va lister ce qu'il y a dans l'exercice LinqEntityFramework :

Remarque : *Fake repository.*
Donne des données pour faire du LinQ dessus (repository sera un objet qui renverra des données à partir d'une base).

Note :
Les méthodes Linq Prennent des fonctions anonymes en paramètre. => indique la déclaration d'une fonction. Ces fonction recoive comme paramètre l'élément courant (les contenues du conteneurs)

Remarque : *First et FirstOrDefault.*
First renvoie l'élément et une erreur si il ne le trouve pas. FirstOrDefault renvoie l'élement ou null si il ne trouve pas

Remarque : *Single et SingleOrDefault.*
// Single renvoie l'élément et une erreur si il ne le trouve pas ou si il y en a plusieur. SingleOrDefault renvoie l'élement ou null si il ne trouve pas ou bien une exception si il en trouve plusieurs

2 Entity Framework

ORM : Mapping Objet relationnel (*object-relational mapping*); Voir : https://fr.wikipedia.org/wiki/Mapping_objet-relationnel. Interface entre la ddb et le code. Donne accès à des listes en base de donnée sur lesquel en va pouvoir venir faire des requêtes.

Note :
Entity Framework ne gère pas les uint.

Dans poec.sql.repository (SqlDbContext). On a besoin des package Microsoft.EntityFrameworkCore et Microsoft.EntityFrameworkCore.SqlServer
Remarque : *Lazi Loading Enabled.*
Chargement différé. permet de faire eleve.classe, ou classe.eleve, pour arriver à eleve.classe.eleve....A creuser... Si activé les table sont récupéré à la volé ce qui mène à plus de requete qu'un chargement au début (il va faire des requetes à des moments non maitrisés). En le désactivant, soit on ne charge pas les données de la table, soit on les charges au moment de la demande via une seule requête.

Remarque : *AutoDetectChangesEnabled.*
Sauvegarde automatique des changements, peut être dangeureux car on peut enregistrer des choses que l'on ne veut pas. On préfère le passer à false pour maitriser la donnée de bout en bout. Entity track tt les changement d'état d'un objet, si autodetectChangeEnable est à false, les changement ne sont plus tracké en permanence mais uniquement explicitement via SaveChange.

Remarque : *DbSet.*
Exécute une requete uniquement au moment ou l'on demande de récupérer un élément. Renvoie la plupart du temps une image de la base de donnée et ne fera la requete qu'au dernier moment. C'est une projection de la base de donnée.

Remarque : *DbContext.*
Objet qui fait le lien avec la base de donnée.

Remarque : *Sur le mapping.*

On peut le faire coté SqlDto (et un peu du DbContext) via des attribut (annotation [blabla] au dessus d'une classe, méthode, property) ou bien directement dans le DbContext (voir DbContext.cs). Il faut faire soit l'un soit l'autre.

Mot clef : **SaveChanges**.

Permet de valider la transaction [qu'on appelle commit](tant que l'on ne fait pas le save change, le DbContext n'applique pas les modification à la BDD) → Principe de transaction (possible aussi en SQL pure). En cas d'erreur le DbContext fait un roll back depuis le dernier `SaveChanges`. `SaveChange` retourne un int correspondant au nombre de ligne affectée par les changement. Cela permet de tester la réussite ou non d'"une opération `Remove` par exemple.

Remarque : *Sur l'autoincrément dans la bdd.*

Dans une BDD, l'autoincrément ne prend pas la dernière ligne mais le dernier élément inséré. Après une supression la base garde quand même en mémoire l'id de l'élément supprimé. Un ajout après cela aura donc un incrément de $+1$ par rapport au dernier supprimé et non pas par rapport à la dernière ligne actuelle.

Remarque : *Sur les Add et Update.*

La bonne pratique est de renvoyer l'objet ajouter/modifier pour tester la nullité de l'objet ou non (et potentiellement vérifier son identité).

Note :

SqlContext.Where() retourne un IQueryable qui a le comportement donnée dans la remarque suivante.

Remarque : *Sur les performance.*

SqlContext.Set<UserSqlDto>().Where(), projette une requete mais ne l'éxécute pas immédiattement. Il faut ensuite faire un .Fisrt, .List, ... pour que la requete soit éxécuté. Cela permet de ne pas faire de requet inutile à chauqe Set ou Where et de ne les éxécuter qu'au besoin

Remarque : *sur le optionBuilder.*

Il est préférable de le sortir du DbContext et le passer en paramètre pour pouvoir découpler la ConnectionString du DbContext.

Remarque : *Sur les UserRepository.*

Il faut une interface sur les XUserRepository pour découpler le UserRepository du type de SQL que l'on utilises. Les fichiers de test et d'utilisation utilise alors un IUserRepository au lieu d'un SqlUserRepository

Mot clef : **Queries option**.

Voir ce lien pour configurer les options proprement à la place de ce que l'on à fait dans SqlDbContext.OnConfiguration.

Note :

Il faut Entity Framework Cor 3.x pour fonctionner avec Wcf et .Net stadart 2.0

3 Divers

Note :
Resharper (jetBrains) : analyse de code, changement dynamique, code helper,... Attention Payant.

Remarque : *Sur les région.*
Se déclare avec des #region #endregion (directive **prépocesseur**).

Remarque : *Sur less CRUD.*
Il faut essayer un maximum de faire les CRUD avec des types primitifs (pour rester le plus générique possible → notion de *scalabilité* du code).

Remarque : *Pattern AAA.*
Arrange, Action, Assert : utiliser pour du test. Evite de recopier du code plusieurs fois. Attention, On peut avoir autant d'arrange et assert que l'on veut mais une seule action (car test unitaire ! ! !).

Remarque : *Bonne pratique.*
Il vaut mieux renvoyer des interfaces si possible (IList au lieu de List).

Note :
On peut utiliser `default` pour forcer une valeur par défaut sur un des paramètres passés lors de l'appel d'une méthode, d'une instanciation ou autre.

Remarque : *Sur les Dto.*
Attention les dto sont réservé à la couche de transfert (communication back et front : communication client et serveur). En BDD on parlera plutôt des Entity (entre le back et le BDD). Exemple : UserEntity au lieu de UserSqlDto.

Remarque : *Select.*
Permet de faire une projection d'un objet vers l'espace des f(objet).

Remarque : *Sur BDD.*
On peut faire miger les base de données. Voir https://docs.microsoft.com/fr-fr/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli.

Remarque : *Sur la migration.*
Attention pour faire une migration il faut passer par un projet en .NET 6 et pas en standart 2.0 (juste pour faire la migration). Il faut juste référencer le projet en .Net 6 pour faire migrer l'ensembel de la solution.

Note :
Il faut probablement utiliser entity framework et par core pour que tt fonctionne avec wcf.

Note :
On peut passer et on doit passer les entité en private/internal mais pas en public. Il faut de plus utiliser des Guards pour faire de la validation (maxlength et compagnie).

Remarque : *Sur le delete.*
On a pas besoin de Get sur l'id. On peut plutôt construire une entité avec juste un id et passer sa

au delete. Cela évite les aller retours pour rien.

Note :
On peut analyser ce que fait EFCore avec les outils de visual studio.

4 Passe de Thomas

Note :
Attention au InMemory → car il n'est ne fait appel au packahe *Relationnal*. Il vaut mieux passer par du SqLite.

Remarque : *Sur le DbContext.*
Le DbContext permet de maintenir le lien entre les objets dans une appli et leur état dans la base de donnée. C'est un point d'entrée à la base de donnée (il a une connection) mais il n'est pas que sa. Le DbContext est un UOW (*Unit of Work*) qui fait le proxy entre la base et les objets.

Remarque : *Sur le principe d'Unit of Work.*
Tant que l'on ne donne pas un ordre d'excécution (SaveChanges pour DbContext) rien ne se passe (Coté base).

Remarque : *Sur l'override du OnConfiguring.*
Cela permet de faire la configuration statique (Fixe). Si l'on veut être un peut plus dynamique il faut passer par les options.

Note :
On peut mettre les DbSet en private set. EF initilisera quand même le DbSet. De plus on ne passe jamais par un = donc le setter n'a pas à être publique. Le stter doit tt de même exister sinon EF ne peut pas initiliser le DbSet. On peut mettre le setter en *init* sinon.

Remarque : *Sur les DbSet.*
Un DbSet est une classe générique qui hérite de IQueryable, qui lui même hérite de IEnumerable. DbSet est donc un type de liste. Il implménete du *Late Exécution* : tant que l'on ne forme pas un aggrégat rien n'est matérialisé.

Remarque : *Sur l'utilisation du DbSet.*
On peut faire des action directement sur le DbContext ou bien sur le DbSet.

Remarque : *Sur les méthodes des Set (ou Context).*
Seules, Add, Delete, ... ne mettent pas à jour le DbContext. Il faut forcément passer par le SaveChanges(). Cela permet d'appliquer un principe transactionnelle : on assure l'intégrité des données.

Remarque : *Sur les constructeurs.*
Les constructeurs par défaut ne absoluement pas obligatoires !

Remarque : *Sur le Set.*
Le Set<T>. Cela permet de renvoyer le DbSet<T> déjà déclaré dans le DbContext. Context.MesClass est équivalent à Context.Set<MaClass> si MesClass est déclaré comme DbSet<MaClass> MesClass {get ; set ;}.

Remarque : *Sur le IDisposable.*

C'est une bonne pratique d'utiliser un using var MonContext = new MonDbContext() (pour pouvoir dispose() dessus.).

Remarque : *Sur la connection à la bdd.*

Ce n'est pas vraiement le DbContext qui se connecte mais le provider (SqlLite, SqlServer...) qui gère la connection. Après la première connection le provider gère un pool de connection qu'il répartit à sa guise par la suite.

Remarque : *Sur les aller retour en base.*

Exemple du Delete. Pour delete on peut d'abord récupérer un objet pour ensuite le delete. Cela fonctionne mais cause deux appel à la base !. On peut faire mieux. En utilisant un constructeur avec un paramètre d'identifiant pour créer une entité avec l'identifiant qui va permettre la suppression. Attention cela peut ne pas fonctionner car la ref des deux objet est différent, et le db context en cours ne pourra pas faire la distinction entre les deux si on utilise le même DbContext pour l'ajout et la suppression. Il lèvera une erreur en disant que l'on essaie d'ajouter un objet qui existe déjà. Pour pouvoir faire sa il faut passer par un DbCOntext qui ne connait pas déjà l'entité à supprimer !

Note :

Sur le comportement par défaut de EFSi l'on appel une property Id ou bien MAClassId, EF considère que cette attribut sera une clef primaire.

Remarque : *Sur la publication d'un build.*

Click droit sur un projet -> publish permet d'ouvrir un utilitaire de publication.

Remarque : *Sur le ClickOnce.*

Permet d'installer un application depuis une page Web (automatiquement).

Remarque : *Sur le publication dans un folder.*

Le Fodler location est l'emplacement ou sera généré le programme. Une fois le profil définie on le verra apparaitre dans l'application. Enfin il n'y a plus qu'à clicker sur publish. Avant sa, dans les settings $\rightarrow$ show all settings, on peut modifier des options qui ne sont pas proposé au début.

Remarque : *Sur le mode de déploiement.*

- Self-contain : on embarque le framework.Net avec l'application. L'utilisateur n'aura pas à réinstaller le framwork pour utiliser l'appli. Permet par exemple de gérer les versions ou de faire du déploiement là ou l'installation d'un framework n'est pas possible.
- Framework-dependant : suppose que l'utilisateur aura un framework d'installer. Permet de cibler AnyCpu (n'importe quel architechture)

Remarque : *Sur le target runtime.*

En Self-Contained il faut le spécifier, le code sera alors dépendant le l'architecture.

Remarque : *Produce single file.*

Permet d'embarquer les dll dans le fichiers .exe. Au final on a plus qu'un seul fichier.

Remarque : *TRIM.*

Permet de réduire la taille du fichier final (en ne choisissant que les partie du framework nécessaires).

Note :
Dans les github Action : Il faut désactiver la télémétrie du dotnet via le paramètre DOTNET_CLI_TELEMETRY qu'il faut passer à 1.

Remarque : *Sur la notion d'artefact.*
C'est la sortie d'un pipeline (une suite d'instruction d'exécution). Aboutit à un .exe, .msi, ...

Remarque : *Sur les .msi.*
Un fichier .msi peut être déployé automatiquement (par rapport à un .exe). Il permet d'utiliser des Group Policy Object (GPO) pour gérer le déploiement.

Remarque : *Sur* ***wix toolset****.*
C'est un outils qui permet de faire les builds via les GitActions.

Remarque : *Sur la commande dotnet publish.*
Permet de publier une application. Attention par défaut la commande prend le profil en cours (souvent Debug donc). Il faut lui passer un fichier de profil en argument pour faire appliquer autre chose que le comportement par défaut. Note : cette commande inclue également la commande dotnet restore. C'est en fait un restore+build+publish

Remarque : *Sur la commande dotnet restore.*
Permet de générer les dépendances (les packages Nugget).

Note :
C'est une bonne pratique de séparer restore, build et publish (via –no-restore dans build puis –no-build dans publish) pour pouvoir savoir plus précisément où sont les problèmes lorsqu'ils surviennent.

Remarque : *Sur la commande dotnet test.*
Permet delancer un jeux de test. Attention les tests sont dépendant de la plateforme. Il faut lui préciser la matrice de runtime.

Remarque : *Sur les fichier .pdb.*
Programm Debug Database : fichiers facultatifs mais qui permettent de faire du debug.

Remarque : *Sur le using.*
Il ne sert que l'orsque l'on a besoin d'utiliser un objet de type IDsposable de manière explicite. En ASP Net ce n'est pas nécessaire. Tt les service passées en Scoped Ou transcient appel également la méthode Dispose. Par contre lorsque l'on utilise un IDisposable en dehors de ce genre de context, la bonne pratique est de toujours utilisé un using pour que la méthode dispose soit appelée en temps voulu (cad après la dernière apparition de l'objet dans le programme).

Remarque : *Sur les IDisposable.*
Lorsque l'on utilise un objet IDsiposable (DbContext, Stream, FileWritter) il faut toujours se demander uqi va gérer le dispose et si personne n'est trouvé il faut le faire soit même en utilisant le using.

Remarque : *Sur la méthode Dispose.*
Elle sert à nettoyer les ressources allouées à un objet. L'objet n'est pas détruit (car les destructeur

n'existent pas en C#) mais est pratiquement inutilisable après l'appel à Dispose().

Remarque : *Sur le destructeur en C#.*
Il est possible de le définir comme en C++ mais on ne peut pas l'appeler explicitement, c'est le garbage collector qui l'appel. voir https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/classes-and-structs/finalizers.

Remarque : *Sur le modelBuilder.Entity.HasData.*
Permet de seeder une bdd c'est à dire la remplir avec des données initiales.

Remarque : *Sur DbContext.Database.EnsureCreated().*
Instencie le provider (SQlserveur, ...) puis passe dans la méthodes OnModelCreating.

Remarque : *Sur le Include et LazyLoading.*
De base Entity Framework fait du Lazy loading, il ne remonte aucune des propriétés associées à un objet via une naviguation (enfin si mais il les affecte null). Pour récupérer l'objet associé on peut faire Contenant.Contenue et à ce moment là une requête sera faite !

Note :
Par défaut sur EF6 le lazy loading est désactivé et on est directement en eager loading.

Remarque : *Sur le Eager Loading.*
Ce qui est remonté après une premièr erequête est définit. L'objet courant est remonté par défaut sans les propriétés de naviguation. Un appel a Contenant.Contenue restera nulle et ne déclenchera pas de requête.

Remarque : *Sur la profondeur des Dto.*
Dans la mesure du possible il vaut mieux avoir des Dtos à plat (1 seul niveau d'indendation) mais dans le cas d'une liste on peut se permettre d'avoir un niveau de plus. On a des truc du genre COntenant prop1, Contenant prop2, contenue prop1, contenue prop2.

Note :
Il est possible de faire remonter le builder.Service (IServiceCollection) de la persistance vers l'API pour ne pas avoir à exposer les enity ou les profils d'Automapper.

Note :
Il faut faire attention à ne pas créer des boucles de tables (cad différent chemin pour atteindre une même donnée).
Remarque : *Sur ICollection.*
Laisse une main un peu plus grande sur les propriétés de naviguations que IEnumerable. Pas vraiement de règles la dessus (choix entre abstraction et utilisation pratique.)

Note :
Il est interessant d'avoir des erreur normalisées par types (dto, entity, ...). Puis avec les Exception Detail on peut filtrer plus en détail. L'idée est de faire un type d'erreur par *couche* du programme.

Remarque : *Sur le ChangeTracker.*
Il vaut mieux finalement ne pas le désactiver. Il permet un meilleur debug (via le DebugView). Cela permet également lors d'un update de faire un update uniquement sur les colonnes qui ont changé et pas sur toutes les colonnes. De plus si on a des indexes ou des *vue* liées à certaine tables, la

base va également devoir tt remettre à jour si l'on change toutes les colonnes d'une entités. Le seul intérêt de le désactiver serait pour faire du patch impliquant beaucoup de changement à la fois.

Remarque : *Sur la migration.*
La migration est spécifique à un provider (SqlServeur, SqLite, ...), il faut une migration par provider.

Remarque : *Sur les chaines de connection.*
Voir : https://www.connectionstrings.com/ pour les différentes chaines de connection nécessaires en fonction des provider

Remarque : *Pour utiliser dotnet dans le gestionnaire de package.*
On commence par faire un `new tool-manifest` qui permet de générer un fichier dans lequel on va venir lister les commandes dont on a besoin. Par la suite `dotnet tool install dotnet-ef` viendra rajouter dotnet ef au fichier *manifest*. Cela permet ensuite de lancer `dotnet ef` dans le gestionnaire de package.

Remarque : *Sur le OnDelete(CascadingRule).*
Permet de cascader ou non la supresion de données.

Remarque : *Sur la montées et descentes de version dans les migrations.*
`dotnet ef migrations remove` pour enlever une migration (avant de l'avoir appliqué !). `dotnet ef database update UneMigration` permet d'update à une certaine migration. Cela permet également des retours en arrière au niveau de l'historique des migrations. Attention lors de ces allez retour on peut perdre des données. Lorsque l'on ajoute une colonne, cette dernière prendra une valeur par défaut que l'on peut déterminer dans la migration (si elle n'est pas autorisée à être nulle).

Remarque : *Sur le rajout de données après migration.*
On peut passer du Sql pure (dans le fichier de Migration) avec l'argument defaultValueSql : "MonScriptSql".

Remarque : *Sur migration scripts.*
Permet de générer des scripts SQL correspondant à la création de la migration.

Remarque : *Sur migration bundle.*
`dotnet ef migration bundle` permet de créer une éxécutable qui permet d'éxécuter la migration. Cela évite d'avoir à déployer une appli avec un user en Admin. Cela évite également de faire l'update directement dans nos programmes. Voir cette article https://devblogs.microsoft.com/dotnet/introducing-devops-friendly-ef-core-migration-bundles/ pour un peu plus de détail et https://docs.microsoft.com/fr-fr/ef/core/managing-schemas/migrations/applying?tabs=dotnet-core-cli (section offres groupées). Lorsque l'on a l'exe il suffit d'excécuter le bundle en lui lui passant la chaine de connection : `.\efbundle.exe –connection 'MaChaineDeConnexion'`. Il est aussi possible de passer le nom de la migration que l'on souhaite appliquées.

Remarque : *Si l'on veut lancer une migration dans un program.*
On peut passer par `DbContext.DataBase.Migrate()` . Attention cette commande est mutuellement exclusive avec la commande de création (`EnsureCreated()`).

Note :

En EF Core 6, la commande `DbContext.DataBase.Migrate()` crée la base de donnée si elle n'existe pas.

Remarque : *Sur l'ajout de données pendant la migrations.*

Il est possible de faire sa en passant par les `migrationBuilder.Operations.Add()`.

Remarque : *Sur la migration.*

On devrait mettre les dossiers de migrations dans un projet séparés. Cela permet par exemple de pouvoir target plusieurs type de bases de données.

Remarque : *Si jamais la création de bundle ne fonctionne pas.*

voir un workaround dotnetefmigrationsbundle--configurationBundle--project./src/Myproject/Myproject.csproj--verbose.

Acronymes

DSP Nom simple. , *Glossaire :* DSP

Glossaire

DSP la description détaillé.