eBook Gratuit

APPRENEZ

Entity Framework

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#entity-framework

Table des matières

À propos 1

Chapitre 1: Démarrer avec Entity Framework 2

Remarques 2

Versions 2

Examples 2

Utiliser Entity Framework à partir de C # (Code First) 2

Installation du package Entity Framework NuGet 3

Qu'est-ce que Entity Framework? 7

Chapitre 2: .t4 templates dans entity-framework 9

Examples 9

Ajouter dynamiquement des interfaces au modèle 9

Ajout de documentation XML aux classes d'entités 9

Chapitre 3: Chargement d'entités liées 11

Remarques 11

Examples 11

Chargement paresseux 11

Chargement rapide 12

Fortement typé. 12

Surcharge de chaîne. 12

Chargement explicite 13

Filtrer les entités associées. 13

Requêtes de projection 13

Chapitre 4: Code cadre d'entité en premier 15

Examples 15

Se connecter à une base de données existante 15

Chapitre 5: Code Entité-cadre Premières migrations 17

Examples 17

Activer les migrations 17

Ajoutez votre première migration 17

Semis de données lors des migrations 19

Utiliser Sql () pendant les migrations 20
Autre usage 21
Faire "Update-Database" dans votre code 21
Première étape du code-cadre de l'entité 22
Chapitre 6: Code First - Fluent API 24
Remarques 24
Examples 24
Modèles de cartographie 24
Première étape: Créer un modèle 24
Etape 2: Créer une classe de mappeur 24
Troisième étape: Ajouter une classe de mappage aux configurations. 26
Clé primaire 26
Clé primaire composite 26
Longueur maximale 27
Propriétés requises (NOT NULL) 27
Expliquer la dénomination de la clé étrangère 28
Chapitre 7: Code First DataAnnotations 29
Remarques 29
Examples 29
Attribut [Key] 29
Attribut [obligatoire] 30
Attributs [MaxLength] et [MinLength] 30
[Range (min, max)] attribut 31
Attribut [DatabaseGenerated] 32
Attribut [NotMapped] 33
Attribut [Table] 34
Attribut [Column] 34
Attribut [Index] 35
Attribut [ForeignKey (string)] 35
[StringLength (int)] attribut 36
Attribut [Horodatage] 37
[ConcurrencyCheck] Attribut 37

[InverseProperty (string)] attribut 38
Attribut [ComplexType] 39
Chapitre 8: Conventions de code premier 41
Remarques 41
Examples 41
Convention clé primaire 41
Suppression de conventions 41
Type de découverte 42
DecimalPropertyConvention 43
Convention Relationnelle 44
Convention sur la clé étrangère 45
Chapitre 9: Entity Framework avec SQLite 47
Introduction 47
Examples 47
Configuration d'un projet pour utiliser Entity Framework avec un fournisseur SQLite 47
Installer les bibliothèques gérées SQLite 47
Y compris la bibliothèque non gérée 48
Modifier le fichier App.config du projet 49
Correctifs requis 49
Ajouter une chaîne de connexion SQLite 49
Votre premier SQLite DbContext 50
Chapitre 10: Entity-Framework avec Postgresql 51
Examples 51
Pré-étapes nécessaires pour utiliser Entity Framework 6.1.3 avec PostgresSql en utilisant 51
Chapitre 11: État de l'entité de gestion 52
Remarques 52
Examples 52
Définition de l'état Ajout d'une entité unique 52
Définition de l'état Ajout d'un graphique d'objet 52
Exemple 53
Chapitre 12: Héritage avec EntityFramework (Code First) 54

Examples 54
Tableau par hiérarchie 54
Tableau par type 55
Chapitre 13: Initialiseurs de base de données 57
Examples 57
CreateDatabaseIfNotExists 57
DropCreateDatabaseIfModelChanges 57
DropCreateDatabaseAlways 57
Initialiseur de base de données personnalisé 57
MigrateDatabaseToLatestVersion 58
Chapitre 14: Meilleures pratiques pour l'entité Framework (Simple & Professional) 59
Introduction 59
Examples 59
1- Entity Framework @ Data layer (Bases) 59
2- Couche Entity Framework @ Business 63
3- Utilisation de la couche Business @ Couche de présentation (MVC) 66
4- Entity Framework @ Unit Test Layer 67
Chapitre 15: Modèles de contraintes 71
Examples 71
Relations un-à-plusieurs 71
Chapitre 16: Première génération de base de données 73
Examples 73
Générer un modèle à partir d'une base de données 73
Ajout d'annotations de données au modèle généré 74
Chapitre 17: Relation de correspondance avec Entity Framework Code First: un à plusieurs e... 77
Introduction 77
Examples 77
Cartographier un à plusieurs 77
Cartographier un à plusieurs: contre la convention 78
Cartographier zéro ou un à plusieurs 80
Plusieurs à plusieurs 80
Plusieurs à plusieurs: personnalisation de la table de jointure 81

Many-to-many: entité de jointure personnalisée 83
Chapitre 18: Relation de mappage avec Entity Framework Code First: One-to-one et variation 86
Introduction 86
Examples 86
Cartographier un à zéro ou un 86
Cartographie individuelle 90
Cartographier un ou un zéro ou un zéro 91
Chapitre 19: Scénarios de cartographie avancés: fractionnement d'entité, fractionnement de 92
Introduction 92
Examples 92
Fractionnement d'entité 92
Fractionnement de table 93
Chapitre 20: Suivi ou non suivi 95
Remarques 95
Examples 95
Suivi des requêtes 95
Requêtes sans suivi 95
Suivi et projections 96
Chapitre 21: Techniques d'optimisation dans EF 97
Examples 97
Utiliser AsNoTracking 97
Chargement des données requises uniquement 97
Exécutez les requêtes dans la base de données lorsque cela est possible, pas en mémoire. 98
Exécuter plusieurs requêtes asynchrones et parallèles 99
Mauvais exemple 99
Bon exemple 99
Désactiver le suivi des modifications et la génération de proxy 99
Travailler avec des entités de stub 100
Chapitre 22: Transactions 102
Examples 102
Database.BeginTransaction () 102

Chapitre 23: Types complexes 103

Examples 103

Code premiers types complexes 103

Crédits 104

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: entity-framework

It is an unofficial and free Entity Framework ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Entity Framework.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Entity Framework

Remarques

Entity Framework (EF) est un mappeur objet-relationnel (ORM) qui permet aux développeurs .NET de travailler avec des données relationnelles à l'aide d'objets spécifiques à un domaine. Cela élimine le besoin de la plupart du code d'accès aux données que les développeurs doivent généralement écrire.

Entity Framework vous permet de créer un modèle en écrivant du code ou en utilisant des boîtes et des lignes dans EF Designer. Ces deux approches peuvent être utilisées pour cibler une base de données existante ou créer une nouvelle base de données.

Entity Framework est le principal ORM fourni par Microsoft pour .NET Framework et la technologie d'accès aux données recommandée par Microsoft.

Versions

Version	Date de sortie
1.0	2008-08-11
4.0	2010-04-12
4.1	2011-04-12
4.1 Mise à jour 1	2011-07-25
4.3.1	2012-02-29
5.0	2012-08-11
6,0	2013-10-17
6.1	2014-03-17
Core 1.0	2016-06-27

Notes de publication: https://msdn.microsoft.com/en-ca/data/jj574253.aspx

Examples

Utiliser Entity Framework à partir de C # (Code First)

Le code vous permet d'abord de créer vos entités (classes) sans utiliser de concepteur graphique

ou de fichier .edmx. Il est nommé *premier code,* parce que vous pouvez créer vos modèles d' *abord* et *Entity Framework* créerez base de données en fonction de correspondances pour vous automatiquement. Vous pouvez également utiliser cette approche avec une base de données existante, qui s'appelle d' *abord le code avec la base de données existante.* Par exemple, si vous souhaitez qu'une table contienne une liste de planètes:

```
public class Planet
{
    public string Name { get; set; }
    public decimal AverageDistanceFromSun { get; set; }
}
```

Maintenant, créez votre contexte qui constitue le pont entre vos classes d'entités et la base de données. Donnez-lui une ou plusieurs propriétés `DbSet<>` :

```
using System.Data.Entity;

public class PlanetContext : DbContext
{
    public DbSet<Planet> Planets { get; set; }
}
```

Nous pouvons l'utiliser en procédant comme suit:

```
using(var context = new PlanetContext())
{
    var jupiter = new Planet
    {
        Name = "Jupiter",
        AverageDistanceFromSun = 778.5
    };

    context.Planets.Add(jupiter);
    context.SaveChanges();
}
```

Dans cet exemple, nous créons une nouvelle `Planet` avec la propriété `Name` avec la valeur `"Jupiter"` et la propriété `AverageDistanceFromSun` avec la valeur `778.5`

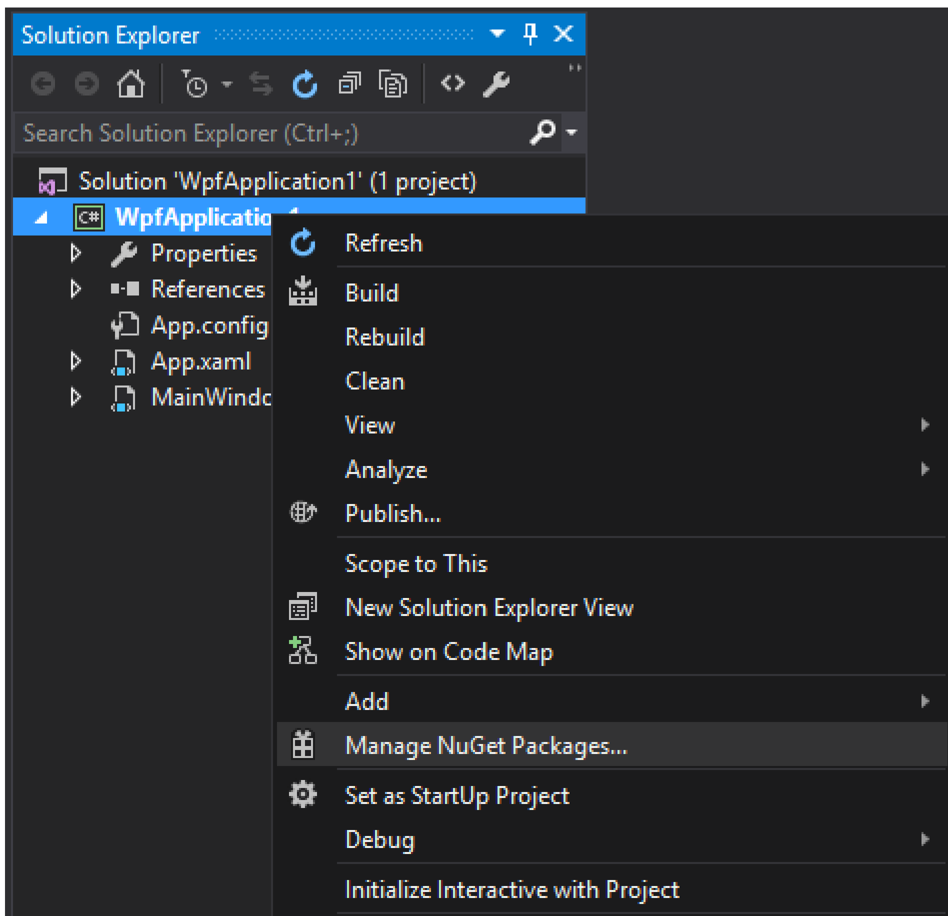
Nous pouvons ensuite ajouter cette `Planet` au contexte en utilisant la `DbSet` `Add()` `DbSet` et valider nos modifications dans la base de données en utilisant la méthode `SaveChanges()` .

Ou nous pouvons récupérer des lignes de la base de données:

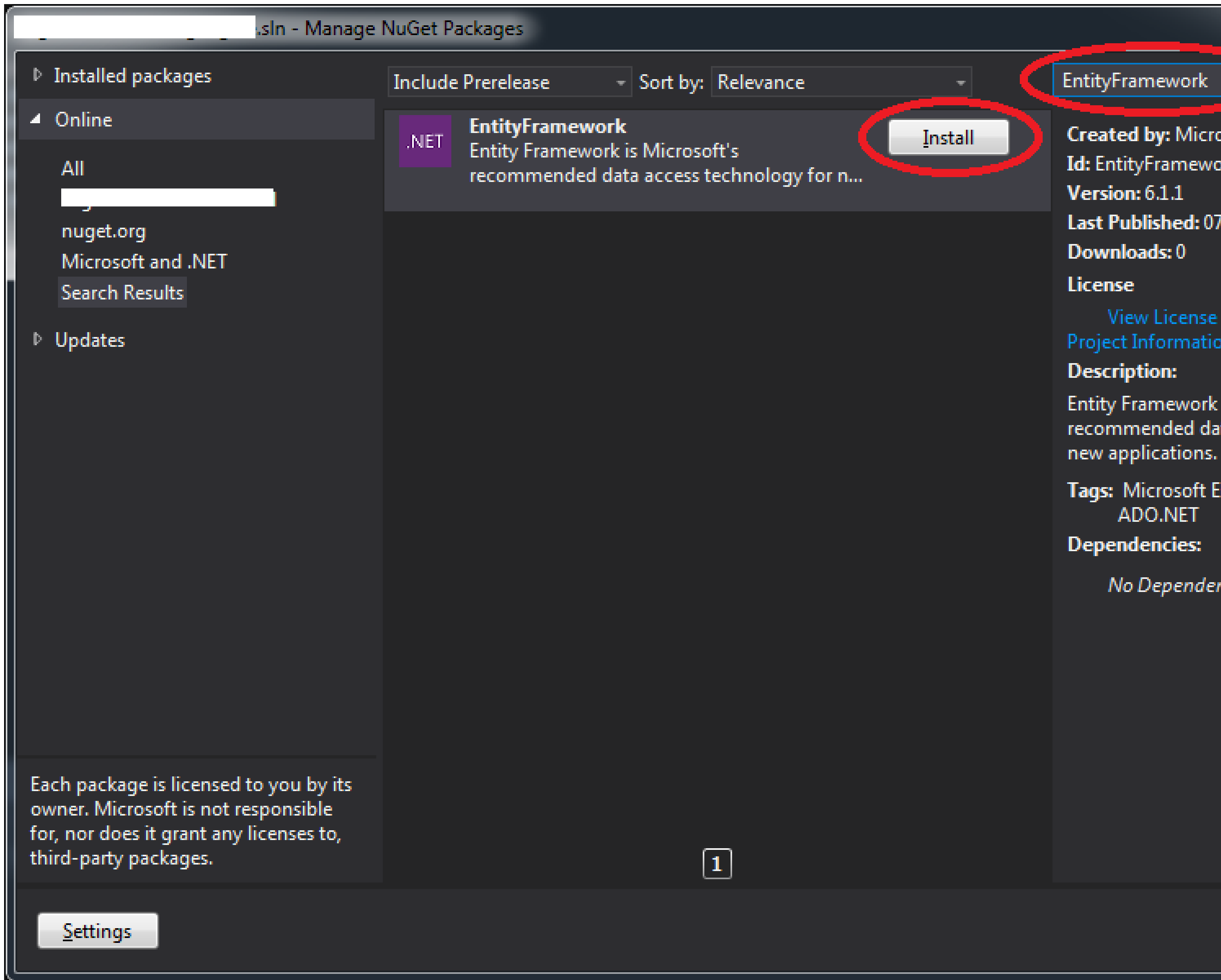
```
using(var context = new PlanetContext())
{
    var jupiter = context.Planets.Single(p => p.Name == "Jupiter");
    Console.WriteLine($"Jupiter is {jupiter.AverageDistanceFromSun} million km from the
sun.");
}
```

Installation du package Entity Framework NuGet

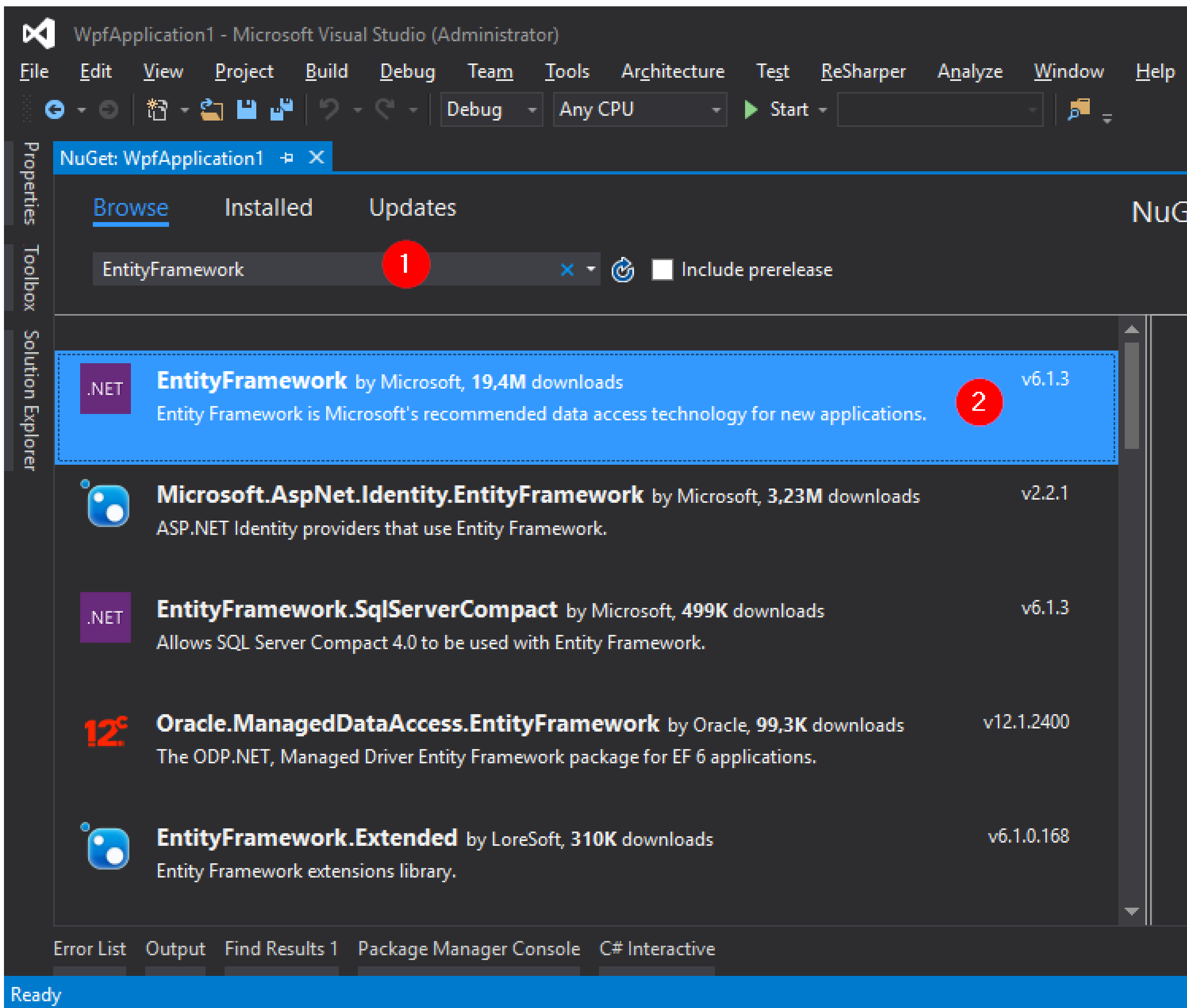
Dans votre Visual Studio, ouvrez la fenêtre **Explorateur de solutions** , puis `cliquez avec` le `bouton droit` sur votre projet, puis sélectionnez *Gérer les packages NuGet* dans le menu:



Dans la fenêtre qui s'ouvre, tapez `EntityFramework` dans le champ de recherche en haut à droite.



Ou, si vous utilisez Visual Studio 2015, vous verrez quelque chose comme ceci:



Puis cliquez sur Installer.

Nous pouvons également installer un framework d'entités à l'aide de la console du gestionnaire de packages. Pour ce faire, ouvrez-le en utilisant le *menu Outils -> NuGet Package Manager -> Console du Gestionnaire de packages,* puis entrez ceci:

```
Install-Package EntityFramework
```

```
Package Manager Console
Package source: nuget.org   Default project: WpfApplication1
Type 'get-help NuGet' to see all available NuGet commands.

PM> Install-Package EntityFramework


Attempting to gather dependency information for package 'EntityFramework.6.1.3' with respect to project
'WpfApplication1', targeting '.NETFramework,Version=v4.6'
Gathering dependency information took 580,37 ms
Attempting to resolve dependencies for package 'EntityFramework.6.1.3' with DependencyBehavior 'Lowest'
Resolving dependency information took 0 ms
Resolving actions to install package 'EntityFramework.6.1.3'
Resolved actions to install package 'EntityFramework.6.1.3'
Retrieving package 'EntityFramework 6.1.3' from 'nuget.org'.
Adding package 'EntityFramework.6.1.3' to folder 'c:\dev\so\WpfApplication1\packages'
Added package 'EntityFramework.6.1.3' to folder 'c:\dev\so\WpfApplication1\packages'
Added package 'EntityFramework.6.1.3' to 'packages.config'
Executing script file 'c:\dev\so\WpfApplication1\packages\EntityFramework.6.1.3\tools\init.ps1'
Executing script file 'c:\dev\so\WpfApplication1\packages\EntityFramework.6.1.3\tools\install.ps1'

Type 'get-help EntityFramework' to see all available Entity Framework commands.
Successfully installed 'EntityFramework 6.1.3' to WpfApplication1
Executing nuget actions took 7,94 sec
Time Elapsed: 00:00:09.3130546
PM>

100 %
```

Cela installera Entity Framework et ajoutera automatiquement une référence à l'assembly dans votre projet.

Qu'est-ce que Entity Framework?

L'écriture et la gestion du code ADO.Net pour l'accès aux données est un travail fastidieux et monotone. Microsoft a fourni une **structure O / RM appelée «Entity Framework»** pour automatiser les activités liées aux bases de données pour votre application.

Le framework d'entité est un framework de mappage objet / relationnel (O / RM). Il s'agit d'une amélioration apportée à ADO.NET qui offre aux développeurs un mécanisme automatisé d'accès et de stockage des données dans la base de données.

Qu'est ce que O / RM?

ORM est un outil de stockage de données à partir d'objets de domaine vers la base de données relationnelle telle que MS SQL Server, de manière automatisée, sans trop de programmation. O / RM comprend trois parties principales:

1. Objets de classe de domaine
2. Objets de base de données relationnelle
3. Informations cartographiques sur la façon dont les objets domaine carte à des objets de base de données relationnelles (tables **ex,** vues et procédures stockées)

ORM nous permet de garder notre conception de base de données séparée de notre conception

de classe de domaine. Cela rend l'application maintenable et extensible. Il automatise également les opérations CRUD standard (Create, Read, Update & Delete) afin que le développeur n'ait pas besoin de l'écrire manuellement.

Lire Démarrer avec Entity Framework en ligne: https://riptutorial.com/fr/entity-framework/topic/815/demarrer-avec-entity-framework

Chapitre 2: .t4 templates dans entity-framework

Examples

Ajouter dynamiquement des interfaces au modèle

Lorsque vous travaillez avec un modèle existant qui est assez gros et qui est régénéré assez souvent dans les cas où une abstraction est nécessaire, il peut être coûteux de contourner manuellement la redécoration du modèle avec des interfaces. Dans de tels cas, on peut vouloir ajouter un comportement dynamique à la génération de modèle.

L'exemple suivant montre comment ajouter automatiquement des interfaces sur des classes ayant des noms de colonnes spécifiques:

Dans votre modèle, allez dans le fichier `.tt` modifiez la méthode `EntityClassOpening` de la manière suivante, cela ajoutera l'interface `IPolicyNumber` sur les entités qui ont `POLICY_NO` colonne `IUniqueId` et `IUniqueId` sur `UNIQUE_ID`

```
public string EntityClassOpening(EntityType entity)
{
    var stringsToMatch = new Dictionary<string,string> { { "POLICY_NO", "IPolicyNumber" }, {
"UNIQUE_ID", "IUniqueId" } };
    return string.Format(
        CultureInfo.InvariantCulture,
        "{0} {1}partial class {2}{3}{4}",
        Accessibility.ForType(entity),
        _code.SpaceAfter(_code.AbstractOption(entity)),
        _code.Escape(entity),
        _code.StringBefore(" : ", _typeMapper.GetTypeName(entity.BaseType)),
        stringsToMatch.Any(o => entity.Properties.Any(n => n.Name == o.Key)) ? " : " +
string.Join(", ", stringsToMatch.Join(entity.Properties, l => l.Key, r => r.Name, (l,r) =>
l.Value)) : string.Empty);
}
```

Ceci est un cas spécifique mais il montre la capacité de pouvoir modifier les modèles `.tt` .

Ajout de documentation XML aux classes d'entités

Sur chaque classe de modèle générée, aucun commentaire de documentation n'est ajouté par défaut. Si vous souhaitez utiliser des commentaires de documentation XML pour chaque classe d'entité générée, recherchez cette partie dans *[nom_modèle]* .tt (*nom_modèle* correspond au nom de fichier EDMX actuel):

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code); // used to write model namespace
```

```
#>
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
```

Vous pouvez ajouter les commentaires de la documentation XML avant d' `UsingDirectives` ligne `UsingDirectives` , comme indiqué dans l'exemple ci-dessous:

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code);
#>
/// <summary>
/// <#=entity.Name#> model entity class.
/// </summary>
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
```

Le commentaire de la documentation générée doit inclure le nom de l'entité, comme indiqué ci-dessous.

```
/// <summary>
/// Example model entity class.
/// </summary>
public partial class Example
{
    // model contents
}
```

Lire .t4 templates dans entity-framework en ligne: https://riptutorial.com/fr/entity-framework/topic/3964/-t4-templates-dans-entity-framework

Chapitre 3: Chargement d'entités liées

Remarques

Si les modèles sont correctement associés, vous pouvez facilement charger les données associées à l'aide d'EntityFramework. Vous avez le choix entre trois options: *chargement différé, chargement rapide* et *chargement explicite* .

Modèles utilisés dans les exemples:

```
public class Company
{
    public int Id { get; set; }
    public string FullName { get; set; }
    public string ShortName { get; set; }

    // Navigation properties
    public virtual Person Founder { get; set; }
    public virtual ICollection<Address> Addresses { get; set; }
}

public class Address
{
    public int Id { get; set; }
    public int CompanyId { get; set; }
    public int CountryId { get; set; }
    public int CityId { get; set; }
    public string Street { get; set; }

    // Navigation properties
    public virtual Company Company { get; set; }
    public virtual Country Country { get; set; }
    public virtual City City { get; set; }
}
```

Examples

Chargement paresseux

Le chargement différé est activé par défaut. Le chargement différé est réalisé en créant des classes proxy dérivées et en remplaçant les produits de navigation virtuels. Le chargement paresseux se produit lorsque la propriété est accédée pour la première fois.

```
int companyId = ...;
Company company = context.Companies
    .First(m => m.Id == companyId);
Person founder = company.Founder; // Founder is loaded
foreach (Address address in company.Addresses)
{
    // Address details are loaded one by one.
}
```

Pour désactiver le chargement paresseux pour des propriétés de navigation spécifiques, supprimez simplement le mot-clé virtuel de la déclaration de propriété:

```
public Person Founder { get; set; } // "virtual" keyword has been removed
```

Si vous souhaitez désactiver complètement le chargement paresseux, vous devez modifier la configuration, par exemple, dans le *constructeur Context* :

```
public class MyContext : DbContext
{
    public MyContext(): base("Name=ConnectionString")
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

Remarque: n'oubliez pas de *désactiver le* chargement paresseux si vous utilisez la sérialisation. Les sérialiseurs accédant à toutes les propriétés, vous les chargez tous depuis la base de données. De plus, vous pouvez exécuter une boucle entre les propriétés de navigation.

Chargement rapide

Le chargement rapide vous permet de charger toutes les entités nécessaires à la fois. Si vous préférez que toutes vos entités travaillent sur un même appel de base de données, alors le *chargement par Eager* est la solution. Il vous permet également de charger plusieurs niveaux.

Vous avez *deux options* pour charger les entités associées, vous pouvez choisir soit des *typologies fortes,* soit *des* surcharges de *chaînes* de la méthode *Include* .

Fortement typé.

```
// Load one company with founder and address details
int companyId = ...;
Company company = context.Companies
    .Include(m => m.Founder)
    .Include(m => m.Addresses)
    .SingleOrDefault(m => m.Id == companyId);

// Load 5 companies with address details, also retrieve country and city
// information of addresses
List<Company> companies = context.Companies
    .Include(m => m.Addresses.Select(a => a.Country));
    .Include(m => m.Addresses.Select(a => a.City))
    .Take(5).ToList();
```

Cette méthode est disponible depuis Entity Framework 4.1. Assurez-vous d'avoir la référence à l' `using System.Data.Entity;` ensemble.

Surcharge de chaîne.

```
// Load one company with founder and address details
int companyId = ...;
Company company = context.Companies
    .Include("Founder")
    .Include("Addresses")
    .SingleOrDefault(m => m.Id == companyId);

// Load 5 companies with address details, also retrieve country and city
// information of addresses
List<Company> companies = context.Companies
    .Include("Addresses.Country");
    .Include("Addresses.City"))
    .Take(5).ToList();
```

Chargement explicite

Après avoir désactivé le *chargement paresseux,* vous pouvez charger des entités paresseusement en appelant explicitement la méthode *Load* pour les entrées. *La référence* est utilisée pour charger des propriétés de navigation uniques, tandis que *Collection* est utilisée pour obtenir des collections.

```
Company company = context.Companies.FirstOrDefault();
// Load founder
context.Entry(company).Reference(m => m.Founder).Load();
// Load addresses
context.Entry(company).Collection(m => m.Addresses).Load();
```

Comme sur *Eager loading,* vous pouvez utiliser les surcharges des méthodes ci-dessus pour charger entiteis par leurs noms:

```
Company company = context.Companies.FirstOrDefault();
// Load founder
context.Entry(company).Reference("Founder").Load();
// Load addresses
context.Entry(company).Collection("Addresses").Load();
```

Filtrer les entités associées.

En utilisant la méthode *Query* , nous pouvons filtrer les entités liées chargées:

```
Company company = context.Companies.FirstOrDefault();
// Load addresses which are in Baku
context.Entry(company)
    .Collection(m => m.Addresses)
    .Query()
    .Where(a => a.City.Name == "Baku")
    .Load();
```

Requêtes de projection

Si l'on a besoin de données associées dans un type dénormalisé, ou par exemple uniquement un sous-ensemble de colonnes, on peut utiliser des requêtes de projection. S'il n'y a aucune raison

d'utiliser un type supplémentaire, il est possible de joindre les valeurs dans un type anonyme .

```
var dbContext = new MyDbContext();
var denormalizedType = from company in dbContext.Company
                       where company.Name == "MyFavoriteCompany"
                       join founder in dbContext.Founder
                       on company.FounderId equals founder.Id
                       select new
                       {
                           CompanyName = company.Name,
                           CompanyId = company.Id,
                           FounderName = founder.Name,
                           FounderId = founder.Id
                       };
```

Ou avec query-syntax:

```
var dbContext = new MyDbContext();
var denormalizedType = dbContext.Company
                       .Join(dbContext.Founder,
                                    c => c.FounderId,
                                    f => f.Id ,
                                    (c, f) => new
                                    {
                                        CompanyName = c.Name,
                                        CompanyId = c.Id,
                                        FounderName = f.Name,
                                        FounderId = f.Id
                                    })
                       .Select(cf => cf);
```

Lire Chargement d'entités liées en ligne: https://riptutorial.com/fr/entity-framework/topic/4678/chargement-d-entites-liees

Chapitre 4: Code cadre d'entité en premier

Examples

Se connecter à une base de données existante

Pour réaliser la tâche la plus simple dans Entity Framework - pour vous connecter à une base de données existante, `ExampleDatabase` sur votre instance locale de MSSQL, vous devez implémenter uniquement deux classes.

La première est la classe d'entité, qui sera mappée à notre table de base de données `dbo.People` .

```
class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
}
```

La classe utilisera les conventions Entity Framework et `dbo.People` table `dbo.People` qui doit avoir la clé primaire `PersonId` et une propriété varchar (max) `FirstName` .

La seconde est la classe de contexte qui dérive de `System.Data.Entity.DbContext` et qui gérera les objets d'entité pendant l'exécution, les dirigera depuis la base de données, gérera la concurrence et les enregistrera dans la base de données.

```
class Context : DbContext
{
    public Context(string connectionString) : base(connectionString)
    {
        Database.SetInitializer<Context>(null);
    }

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

Veuillez noter que dans le constructeur de notre contexte, nous devons définir l'initialiseur de base de données sur null - nous ne voulons pas qu'Entity Framework crée la base de données, nous voulons simplement y accéder.

Maintenant , vous êtes en mesure de manipuler les données de cette table, par exemple , changer le `FirstName` de la première personne dans la base de données à partir d' une application console comme ceci:

```
class Program
{
    static void Main(string[] args)
    {
        using (var ctx = new Context("DbConnectionString"))
        {
            var firstPerson = ctx.People.FirstOrDefault();
```

```
                if (firstPerson != null) {
                    firstPerson.FirstName = "John";
                    ctx.SaveChanges();
                }
            }
        }
    }
```

Dans le code ci-dessus, nous avons créé une instance de Context avec un argument "DbConnectionString". Cela doit être spécifié dans notre fichier `app.config` comme ceci:

```
<connectionStrings>
  <add  name="DbConnectionString"
  connectionString="Data Source=.;Initial Catalog=ExampleDatabase;Integrated Security=True"
  providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Lire Code cadre d'entité en premier en ligne: https://riptutorial.com/fr/entity-framework/topic/5337/code-cadre-d-entite-en-premier

Chapitre 5: Code Entité-cadre Premières migrations

Examples

Activer les migrations

Pour activer Code First Migrations dans la structure d'entité, utilisez la commande

```
Enable-Migrations
```

sur la *console du gestionnaire de packages* .

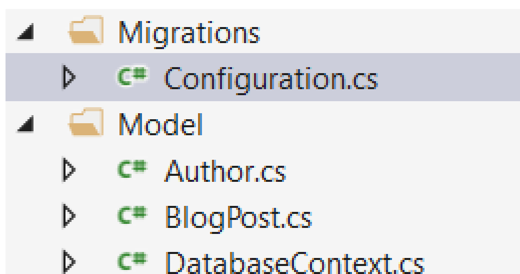
Vous devez avoir une implémentation `DbContext` valide contenant vos objets de base de données gérés par EF. Dans cet exemple, le contexte de la base de données contiendra les objets `BlogPost` et `Author` :

```
internal class DatabaseContext: DbContext
{
    public DbSet<Author> Authors { get; set; }

    public DbSet<BlogPost> BlogPosts { get; set; }
}
```

Après avoir exécuté la commande, la sortie suivante doit apparaître:

```
PM> Enable-Migrations
Checking if the context targets an existing database...
Code First Migrations enabled for project <YourProjectName>.
PM>
```

De plus, un nouveau dossier `Migrations` devrait apparaître avec un seul fichier `Configuration.cs` intérieur:



L'étape suivante consisterait à créer votre premier script de migration de base de données qui créera la base de données initiale (voir l'exemple suivant).

Ajoutez votre première migration

Après avoir activé les migrations (veuillez vous reporter à cet exemple), vous pouvez désormais

créer votre première migration contenant une création initiale de toutes les tables, index et connexions de base de données.

Une migration peut être créée en utilisant la commande

```
Add-Migration <migration-name>
```

Cette commande créera une nouvelle classe contenant deux méthodes, `Up` et `Down` , utilisées pour appliquer et supprimer la migration.

Maintenant, appliquez la commande basée sur l'exemple ci-dessus pour créer une migration appelée *Initial* :

```
PM> Add-Migration Initial
Scaffolding migration 'Initial'.
The Designer Code for this migration file includes a snapshot of your current Code
First model. This snapshot is used to calculate the changes to your model when you
scaffold the next migration. If you make additional changes to your model that you
want to include in this migration, then you can re-scaffold it by running
'Add-Migration Initial' again.
```

Un nouveau fichier d' *horodatage* _Initial.cs est créé (seuls les éléments importants sont affichés ici):

```
public override void Up()
{
   CreateTable(
       "dbo.Authors",
        c => new
           {
               AuthorId = c.Int(nullable: false, identity: true),
               Name = c.String(maxLength: 128),
           })
        .PrimaryKey(t => t.AuthorId);

    CreateTable(
       "dbo.BlogPosts",
       c => new
           {
                Id = c.Int(nullable: false, identity: true),
                Title = c.String(nullable: false, maxLength: 128),
                Message = c.String(),
                Author_AuthorId = c.Int(),
            })
        .PrimaryKey(t => t.Id)
        .ForeignKey("dbo.Authors", t => t.Author_AuthorId)
        .Index(t => t.Author_AuthorId);
}

public override void Down()
{
    DropForeignKey("dbo.BlogPosts", "Author_AuthorId", "dbo.Authors");
    DropIndex("dbo.BlogPosts", new[] { "Author_AuthorId" });
    DropTable("dbo.BlogPosts");
    DropTable("dbo.Authors");
}
```

Comme vous pouvez le voir, dans la méthode `Up()` deux tables `Authors` et `BlogPosts` sont créées et les champs sont créés en conséquence. De plus, la relation entre les deux tables est créée en ajoutant le champ `Author_AuthorId` . De l'autre côté, la méthode `Down()` tente d'inverser les activités de migration.

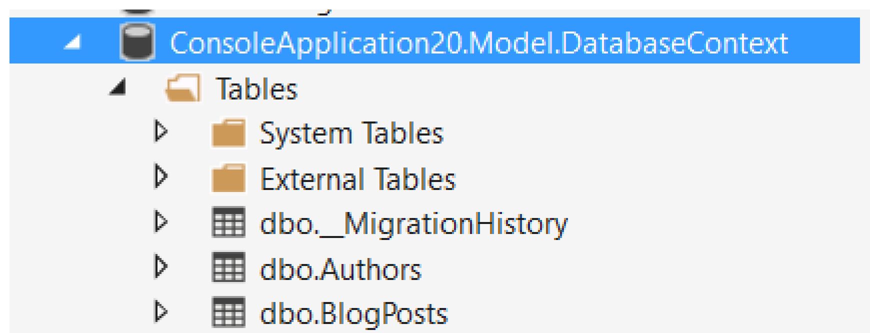
Si vous êtes sûr de votre migration, vous pouvez appliquer la migration à la base de données en utilisant la commande:

```
Update-Database
```

Toutes les migrations en attente (dans ce cas, la migration *initiale*) sont appliquées à la base de données et ensuite la méthode d'amorçage est appliquée (l'exemple approprié)

```
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target
database.
Applying explicit migrations: [201609302203541_Initial].
Applying explicit migration: 201609302203541_Initial.
Running Seed method.
```

Vous pouvez voir les résultats des activités dans l'explorateur SQL:



Pour les commandes `Add-Migration` et `Update-Database` plusieurs options sont disponibles et peuvent être utilisées pour modifier les activités. Pour voir toutes les options, veuillez utiliser

```
get-help Add-Migration
```

et

```
get-help Update-Database
```

Semis de données lors des migrations

Après avoir activé et créé des migrations, il peut être nécessaire de remplir ou de migrer initialement les données de votre base de données. Il y a plusieurs possibilités mais pour les migrations simples, vous pouvez utiliser la méthode 'Seed ()' dans le fichier Configuration créé après l'appel de `enable-migrations` .

La fonction `Seed()` récupère un contexte de base de données uniquement en tant que paramètre et vous pouvez effectuer des opérations EF dans cette fonction:

```
protected override void Seed(Model.DatabaseContext context);
```

Vous pouvez effectuer tous les types d'activités à l'intérieur de `Seed()` . En cas de défaillance, la transaction complète (même les correctifs appliqués) est annulée.

Un exemple de fonction qui crée des données uniquement si une table est vide peut ressembler à ceci:

```
protected override void Seed(Model.DatabaseContext context)
{
    if (!context.Customers.Any()) {
        Customer c = new Customer{ Id = 1, Name = "Demo" };
        context.Customers.Add(c);
        context.SaveData();
    }
}
```

Une fonctionnalité intéressante fournie par les développeurs EF est la méthode d'extension `AddOrUpdate()` . Cette méthode permet de mettre à jour des données en fonction de la clé primaire ou d'insérer des données si elles n'existent pas déjà (l'exemple est tiré du code source généré de Configuration.cs):

```
protected override void Seed(Model.DatabaseContext context)
{
    context.People.AddOrUpdate(
      p => p.FullName,
      new Person { FullName = "Andrew Peters" },
      new Person { FullName = "Brice Lambson" },
      new Person { FullName = "Rowan Miller" }
    );
}
```

> Sachez que `Seed()` est appelé après l'application du **dernier** correctif. Si vous avez besoin de migrer ou de générer des données pendant les correctifs, d'autres approches doivent être utilisées.

Utiliser Sql () pendant les migrations

Par exemple: Vous allez migrer une colonne existante de non requise à obligatoire. Dans ce cas, vous devrez peut-être renseigner des valeurs par défaut dans votre migration pour les lignes où les champs modifiés sont réellement `NULL` . Si la valeur par défaut est simple (par exemple "0"), vous pouvez utiliser une propriété `default` ou `defaultSql` dans votre définition de colonne. Si ce n'est pas si facile, vous pouvez utiliser la fonction `Sql()` dans les fonctions membres `Up()` ou `Down()` de vos migrations.

Voici un exemple. En supposant une classe *auteur* qui contient une adresse e-mail dans le cadre de l'ensemble de données. Nous décidons maintenant d'avoir l'adresse e-mail comme champ obligatoire. Pour migrer des colonnes existantes, l'entreprise a la *bonne* idée de créer des adresses e-mail factices telles que nom `fullname@example.com` , où nom complet est le nom complet des auteurs sans espaces. L'ajout de l'attribut `[Required]` au champ `Email` créerait la migration

suivante:

```
public partial class AuthorsEmailRequired : DbMigration
{
    public override void Up()
    {
        AlterColumn("dbo.Authors", "Email", c => c.String(nullable: false, maxLength: 512));
    }

    public override void Down()
    {
        AlterColumn("dbo.Authors", "Email", c => c.String(maxLength: 512));
    }
}
```

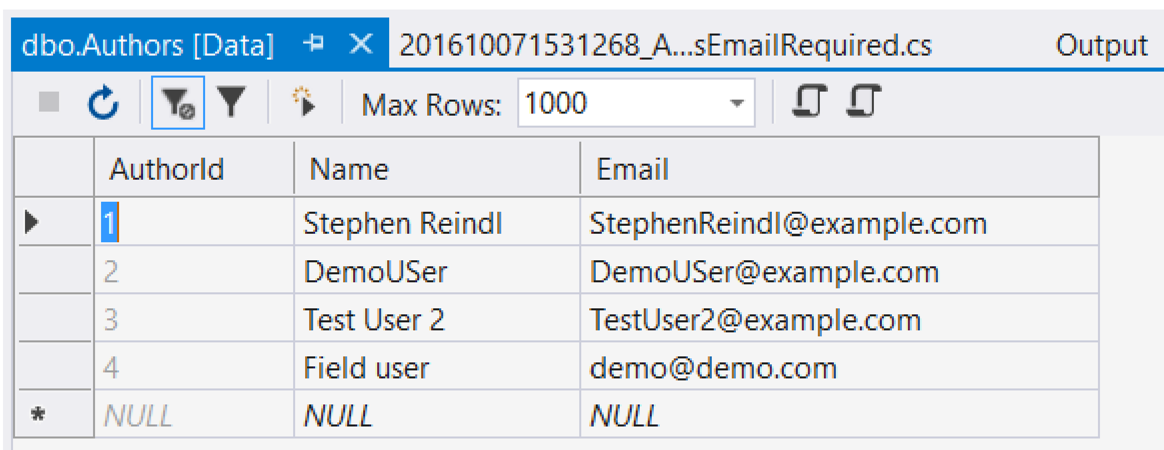
Cela échouerait si certains champs NULL se trouvaient dans la base de données:

> Impossible d'insérer la valeur NULL dans la colonne "Email", table "App.Model.DatabaseContext.dbo.Authors"; colonne n'autorise pas les valeurs NULL. UPDATE échoue.

L'ajout de ce qui suit comme **avant** la commande `AlterColumn` aidera:

```
Sql(@"Update dbo.Authors
    set Email = REPLACE(name, ' ', '') + N'@example.com'
    where Email is null");
```

L'appel `update-database` réussit et la table ressemble à ceci (exemple de données affiché):



	AuthorId	Name	Email
▶	1	Stephen Reindl	StephenReindl@example.com
	2	DemoUSer	DemoUSer@example.com
	3	Test User 2	TestUser2@example.com
	4	Field user	demo@demo.com
*	NULL	NULL	NULL

Autre usage

Vous pouvez utiliser la fonction `Sql()` pour tous les types d'activités DML et DDL dans votre base de données. Il est exécuté dans le cadre de la transaction de migration; Si le SQL échoue, la migration complète échoue et une restauration est effectuée.

Faire "Update-Database" dans votre code

Les applications exécutées dans des environnements autres que de développement nécessitent

souvent des mises à jour de la base de données. Après avoir utilisé la commande `Add-Migration` pour créer vos correctifs de base de données, il est nécessaire d'exécuter les mises à jour sur d'autres environnements, puis sur l'environnement de test.

Les défis généralement rencontrés sont les suivants:

- aucun Visual Studio n'est installé sur les environnements de production et
- pas de connexions autorisées aux environnements de connexion / client dans la vie réelle.

Une solution de contournement est la séquence de code suivante qui vérifie les mises à jour à effectuer et les exécute dans l'ordre. Veuillez vous assurer que les transactions et la gestion des exceptions sont correctes pour vous assurer qu'aucune donnée ne soit perdue en cas d'erreur.

```
void UpdateDatabase(MyDbConfiguration configuration) {
   DbMigrator dbMigrator = new DbMigrator( configuration);
   if ( dbMigrator.GetPendingMigrations().Any() )
   {
      // there are pending migrations run the migration job
      dbMigrator.Update();
   }
}
```

où `MyDbConfiguration` est votre configuration de migration quelque part dans vos sources:

```
public class MyDbConfiguration : DbMigrationsConfiguration<ApplicationDbContext>
```

Première étape du code-cadre de l'entité

1. Créez une application console.
2. Installez le package nuget EntityFramework en exécutant `Install-Package EntityFramework` dans "Package Manager Console"
3. Ajoutez votre chaîne de connexion dans le fichier app.config, il est important d'inclure `providerName="System.Data.SqlClient"` dans votre connexion.
4. Créez une classe publique comme vous le souhaitez, une chose comme " `Blog` "
5. Créez votre ContextClass qui hérite de DbContext, quelque chose comme " `BlogContext` "
6. Définissez une propriété dans votre contexte de type DbSet, quelque chose comme ceci:

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

    public string Name { get; set; }
}
public class BlogContext: DbContext
{
    public BlogContext(): base("name=Your_Connection_Name")
    {
    }

    public virtual DbSet<Blog> Blogs{ get; set; }
}
```

7. Il est important de passer le nom de la connexion dans le constructeur (ici Your_Connection_Name)
8. Dans la console du gestionnaire de packages, exécutez la commande `Enable-Migration` , cela créera un dossier de migration dans votre projet.
9. Exécutez la commande `Add-Migration Your_Arbitrary_Migraiton_Name` , cela créera une classe de migration dans le dossier migrations avec deux méthodes Up () et Down ()
10. Exécuter la commande `Update-Database` afin de créer une base de données avec une table de blog

Lire Code Entité-cadre Premières migrations en ligne: https://riptutorial.com/fr/entity-framework/topic/7157/code-entite-cadre-premieres-migrations

Chapitre 6: Code First - Fluent API

Remarques

Il existe deux manières générales de spécifier HOW Entity Framework qui mappera les classes POCO aux tables de base de données, aux colonnes, etc.: **Annotations de données** et **API Fluent** .

Bien que les annotations de données soient simples à lire et à comprendre, elles manquent de certaines fonctionnalités, telles que la spécification du comportement "Cascade on Delete" pour une entité. D'autre part, l'API Fluent est un peu plus complexe à utiliser, mais fournit un ensemble de fonctionnalités beaucoup plus avancé.

Examples

Modèles de cartographie

EntityFramewok Fluent API est un moyen puissant et élégant de mapper vos modèles de domaine en **code premier** sur une base de données sous-jacente. Cela peut également être utilisé avec *code-first avec la base de données existante* . Vous avez deux options lorsque vous utilisez l' *API Fluent* : vous pouvez mapper directement vos modèles sur la méthode *OnModelCreating* ou vous pouvez créer des classes de mappeur qui héritent de *EntityTypeConfiguration* , puis ajouter ces modèles à *modelBuilder* sur la méthode *OnModelCreating* . *La deuxième* option est celle que je préfère et je vais vous en donner un exemple.

Première étape: Créer un modèle

```
public class Employee
{
    public int Id { get; set; }
    public string Surname { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public short Age { get; set; }
    public decimal MonthlySalary { get; set; }

    public string FullName
    {
        get
        {
            return $"{Surname} {FirstName} {LastName}";
        }
    }
}
```

Etape 2: Créer une classe de mappeur

```
public class EmployeeMap
    : EntityTypeConfiguration<Employee>
{
    public EmployeeMap()
    {
        // Primary key
        this.HasKey(m => m.Id);

        this.Property(m => m.Id)
            .HasColumnType("int")
            .HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);

        // Properties
        this.Property(m => m.Surname)
            .HasMaxLength(50);

        this.Property(m => m.FirstName)
            .IsRequired()
            .HasMaxLength(50);

        this.Property(m => m.LastName)
            .HasMaxLength(50);

        this.Property(m => m.Age)
            .HasColumnType("smallint");

        this.Property(m => m.MonthlySalary)
            .HasColumnType("number")
            .HasPrecision(14, 5);

        this.Ignore(m => m.FullName);

        // Table & column mappings
        this.ToTable("TABLE_NAME", "SCHEMA_NAME");
        this.Property(m => m.Id).HasColumnName("ID");
        this.Property(m => m.Surname).HasColumnName("SURNAME");
        this.Property(m => m.FirstName).HasColumnName("FIRST_NAME");
        this.Property(m => m.LastName).HasColumnName("LAST_NAME");
        this.Property(m => m.Age).HasColumnName("AGE");
        this.Property(m => m.MonthlySalary).HasColumnName("MONTHLY_SALARY");
    }
}
```

Expliquons les cartographies:

- **HasKey** - définit la clé primaire. *Les clés primaires composites* peuvent également être utilisées. Par exemple: *this.HasKey (m => new {m.DepartmentId, m.PositionId})* .
- **Propriété** - permet de configurer les propriétés du modèle.
- **HasColumnType** - spécifie le type de colonne de niveau base de données. Veuillez noter que cela peut être différent pour différentes bases de données comme Oracle et MS SQL.
- **HasDatabaseGeneratedOption** - spécifie si la propriété est calculée au niveau de la base de données. Les PK numériques sont *DatabaseGeneratedOption.Identity* par défaut, vous devez spécifier *DatabaseGeneratedOption.None* si vous ne le souhaitez pas.
- **HasMaxLength** - limite la longueur de la chaîne.
- **IsRequired** - marque la propriété comme requis.
- **HasPrecision** - nous permet de spécifier la précision pour les décimales.
- **Ignorer** - Ignore complètement la propriété et ne la mappe pas à la base de données. Nous

avons ignoré FullName, car nous ne voulons pas que cette colonne figure dans notre table.
- **ToTable** - spécifiez le nom de la table et le nom du schéma (facultatif) pour le modèle.
- **HasColumnName** - relie la propriété au nom de la colonne. Cela n'est pas nécessaire lorsque les noms de propriété et les noms de colonne sont identiques.

Troisième étape: Ajouter une classe de mappage aux configurations.

Nous devons dire à EntityFramework d'utiliser notre classe de mappeur. Pour ce faire, nous devons l’ajouter à *modelBuilder.Configurations* sur la méthode *OnModelCreating* :

```
public class DbContext()
    : base("Name=DbContext")
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Configurations.Add(new EmployeeMap());
    }
}
```

Et c'est tout. Nous sommes tous prêts à partir.

Clé primaire

En utilisant la méthode .HasKey (), une propriété peut être explicitement configurée en tant que clé primaire de l'entité.

```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>().HasKey(p => p.PersonKey);
    }
}
```

Clé primaire composite

En utilisant la méthode .HasKey (), un ensemble de propriétés peut être explicitement configuré en tant que clé primaire composite de l'entité.

```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
```

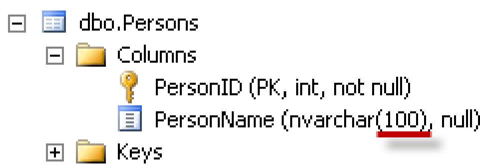
```
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>().HasKey(p => new { p.FirstName, p.LastName });
    }
}
```

Longueur maximale

En utilisant la méthode .HasMaxLength (), le nombre maximal de caractères peut être configuré pour une propriété.

```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>()
                    .Property(t => t.Name)
                    .HasMaxLength(100);
    }
}
```

La colonne résultante avec la longueur de colonne spécifiée:



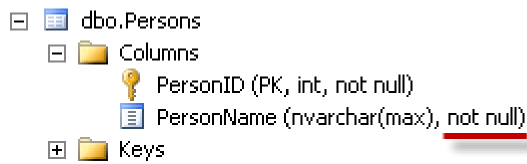
Propriétés requises (NOT NULL)

En utilisant la méthode .IsRequired (), les propriétés peuvent être spécifiées comme obligatoires, ce qui signifie que la colonne aura une contrainte NOT NULL.

```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..
```

```
        modelBuilder.Entity<Person>()
                    .Property(t => t.Name)
                    .IsRequired();
    }
}
```

La colonne résultante avec la contrainte NOT NULL:



Expliquer la dénomination de la clé étrangère

Lorsqu'une propriété de navigation existe sur un modèle, Entity Framework crée automatiquement une colonne de clé étrangère. Si un nom de clé étrangère spécifique est souhaité, mais qu'il ne figure pas en tant que propriété dans le modèle, il peut être défini explicitement à l'aide de l'API Fluent. En utilisant la méthode `Map` lors de l'établissement de la relation avec la clé étrangère, tout nom unique peut être utilisé pour les clés étrangères.

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

public class Employee
{
    property int Id { get; set; }
    property Company Employer { get; set; }
}

public class EmployeeContext : DbContext
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Employee>()
                    .HasRequired(x => x.Employer)
                    .WithRequiredDependent()
                    .Map(m => m.MapKey("EmployerId"));
    }
}
```

Après avoir spécifié la relation, la méthode `Map` permet de définir explicitement le nom de la clé étrangère en exécutant `MapKey` . Dans cet exemple, ce qui aurait abouti à un nom de colonne de Employer_Id est désormais EmployerId.

Lire Code First - Fluent API en ligne: https://riptutorial.com/fr/entity-framework/topic/4530/code-first---fluent-api

Chapitre 7: Code First DataAnnotations

Remarques

Entity Framework Code-First fournit un ensemble d'attributs DataAnnotation, que vous pouvez appliquer à vos classes et propriétés de domaine. Les attributs DataAnnotation remplacent les conventions Code-First par défaut.

1. **System.ComponentModel.DataAnnotations** inclut des attributs qui ont un impact sur la nullité ou la taille de la colonne.
2. **L'** espace de noms **System.ComponentModel.DataAnnotations.Schema** inclut des attributs qui ont un impact sur le schéma de la base de données.

Remarque: DataAnnotations ne vous donne qu'un sous-ensemble d'options de configuration. Fluent API fournit un ensemble complet d'options de configuration disponibles dans Code-First.

Examples

Attribut [Key]

Key est un champ dans une table qui identifie de manière unique chaque ligne / enregistrement dans une table de base de données.

Utilisez cet attribut pour **remplacer la convention Code-First par défaut** . Si elle est appliquée à une propriété, elle sera utilisée comme **colonne de clé primaire** pour cette classe.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    [Key]
    public int PersonKey { get; set; }        // <- will be used as primary key

    public string PersonName { get; set; }
}
```

Si une clé primaire composite est requise, l'attribut [Key] peut également être ajouté à plusieurs propriétés. L'ordre des colonnes dans la clé composite doit être fourni sous la forme [**Key, Column (Order = x)]** .

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    [Key, Column(Order = 0)]
    public int PersonKey1 { get; set; }    // <- will be used as part of the primary key

    [Key, Column(Order = 1)]
    public int PersonKey2 { get; set; }    // <- will be used as part of the primary key
```

```
    public string PersonName { get; set; }
}
```

Sans l'attribut [Key] , EntityFramework utilisera la convention par défaut qui consiste à utiliser la propriété de la classe en tant que clé primaire nommée "Id" ou "{ClassName} Id".

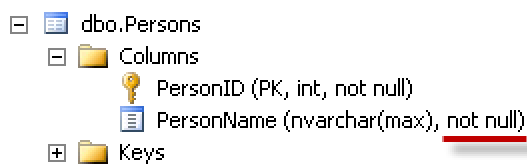
```
public class Person
{
    public int PersonID { get; set; }        // <- will be used as primary key

    public string PersonName { get; set; }
}
```

Attribut [obligatoire]

Appliquée à une propriété d'une classe de domaine, la base de données créera une colonne NOT NULL.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    public int PersonID { get; set; }

    [Required]
    public string PersonName { get; set; }
}
```

La colonne résultante avec la contrainte NOT NULL:



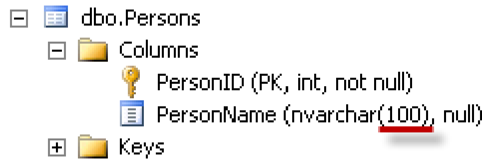
Remarque: il peut également être utilisé avec asp.net-mvc comme attribut de validation.

Attributs [MaxLength] et [MinLength]

L' attribut **[MaxLength (int)]** peut être appliqué à une propriété de type chaîne ou tableau d'une classe de domaine. Entity Framework définira la taille d'une colonne sur la valeur spécifiée.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    public int PersonID { get; set; }

    [MinLength(3), MaxLength(100)]
    public string PersonName { get; set; }
}
```

La colonne résultante avec la longueur de colonne spécifiée:



L'attribut [MinLength (int)] est un attribut de validation, il n'affecte pas la structure de la base de données. Si nous essayons d'insérer / mettre à jour une personne avec PersonName avec une longueur inférieure à 3 caractères, cette validation échouera. Nous allons obtenir une `DbUpdateConcurrencyException` que nous devrons gérer.

```
using (var db = new ApplicationDbContext())
{
     db.Staff.Add(new Person() { PersonName = "ng" });
     try
     {
          db.SaveChanges();
     }
     catch (DbEntityValidationException ex)
     {
          //ErrorMessage = "The field PersonName must be a string or array type with a minimum
length of '3'."
     }
}
```

Les attributs **[MaxLength]** et **[MinLength]** peuvent également être utilisés avec asp.net-mvc comme attribut de validation.

[Range (min, max)] attribut

Spécifie une plage numérique minimale et maximale pour une propriété

```
using System.ComponentModel.DataAnnotations;

public partial class Enrollment
{
    public int EnrollmentID { get; set; }

    [Range(0, 4)]
    public Nullable<decimal> Grade { get; set; }
}
```

Si nous essayons d'insérer / mettre à jour une note avec une valeur hors limites, cette validation échouera. Nous allons obtenir une `DbUpdateConcurrencyException` que nous devrons gérer.

```
using (var db = new ApplicationDbContext())
{
    db.Enrollments.Add(new Enrollment() { Grade = 1000 });

    try
    {
        db.SaveChanges();
    }
```

```
    catch (DbEntityValidationException ex)
    {
        // Validation failed for one or more entities
    }
}
```

Il peut également être utilisé avec asp.net-mvc comme attribut de validation.

Résultat:

Grade 4.01 The field Grade must be between 0 and 4.

Attribut [DatabaseGenerated]

Spécifie comment la base de données génère des valeurs pour la propriété. Il y a trois valeurs possibles:

1. `None` spécifie que les valeurs ne sont pas générées par la base de données.
2. `Identity` spécifie que la colonne est une colonne d'identité , généralement utilisée pour les clés primaires entières.
3. `Computed` spécifie que la base de données génère la valeur pour la colonne.

Si la valeur est différente de `None` , Entity Framework ne remet pas les modifications apportées à la propriété dans la base de données.

Par défaut (basé sur `StoreGeneratedIdentityKeyConvention`), une propriété de clé entière sera traitée comme une colonne d'identité. Pour remplacer cette convention et la forcer à être traitée comme une colonne sans identité, vous pouvez utiliser l'attribut `DatabaseGenerated` avec la valeur `None` .

```
using System.ComponentModel.DataAnnotations.Schema;

public class Foo
{
    [Key]
    public int Id { get; set; } // identity (auto-increment) column
}

public class Bar
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int Id { get; set; } // non-identity column
}
```

Le code SQL suivant crée une table avec une colonne calculée:

```
CREATE TABLE [Person] (
    Name varchar(100) PRIMARY KEY,
    DateOfBirth Date NOT NULL,
    Age AS DATEDIFF(year, DateOfBirth, GETDATE())
```

```
)
GO
```

Pour créer une entité pour représenter les enregistrements du tableau ci-dessus, vous devez utiliser l'attribut `DatabaseGenerated` avec la valeur `Computed` .

```
[Table("Person")]
public class Person
{
    [Key, StringLength(100)]
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public int Age { get; set; }
}
```

Attribut [NotMapped]

Selon la convention Code-First, Entity Framework crée une colonne pour chaque propriété publique d'un type de données pris en charge et dispose à la fois d'un getter et d'un setter. **L'** annotation **[NotMapped]** doit être appliquée à toutes les propriétés pour lesquelles nous ne voulons **PAS** une colonne dans une table de base de données.

Un exemple de propriété que nous pourrions ne pas vouloir stocker dans la base de données est le nom complet de l'étudiant, basé sur son prénom et son nom. Cela peut être calculé à la volée et il n'est pas nécessaire de le stocker dans la base de données.

```
public string FullName => string.Format("{0} {1}", FirstName, LastName);
```

La propriété "FullName" n'a qu'un getter et aucun setter. Par défaut, Entity Framework **ne** créera **PAS de** colonne pour cela.

Un autre exemple de propriété que nous pourrions ne pas vouloir stocker dans la base de données est le "AverageGrade" d'un étudiant. Nous ne voulons pas obtenir le AverageGrade à la demande; au lieu de cela, nous pourrions avoir une routine ailleurs qui le calcule.

```
[NotMapped]
public float AverageGrade { set; get; }
```

Le "AverageGrade" doit être marqué **[NotMapped]** , sinon Entity Framework créera une colonne pour cela.

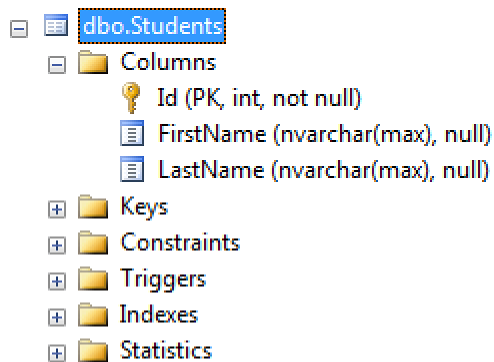
```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    public int Id { set; get; }

    public string FirstName { set; get; }

    public string LastName { set; get; }
```

```
    public string FullName => string.Format("{0} {1}", FirstName, LastName);

    [NotMapped]
    public float AverageGrade { set; get; }
}
```

Pour l'entité ci-dessus, nous verrons à l'intérieur de `DbMigration.cs`

```
CreateTable(
    "dbo.Students",
     c => new
         {
             Id = c.Int(nullable: false, identity: true),
             FirstName = c.String(),
             LastName = c.String(),
         })
     .PrimaryKey(t => t.Id);
```

et dans SQL Server Management Studio



Attribut [Table]

```
[Table("People")]
public class Person
{
    public int PersonID { get; set; }
    public string PersonName { get; set; }
}
```

Indique à Entity Framework d'utiliser un nom de table spécifique au lieu d'en générer un (c.-à-d. Une `Person` ou des `Persons`)

Nous pouvons également spécifier un schéma pour la table en utilisant l'attribut [Table]

```
[Table("People", Schema = "domain")]
```

Attribut [Column]

```
public class Person
{
```

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

    [Column("NameOfPerson")]
    public string PersonName { get; set; }
}
```

Indique à Entity Framework d'utiliser un nom de colonne spécifique à la place en utilisant le nom de la propriété. Vous pouvez également spécifier le type de données de la base de données et l'ordre de la colonne dans la table:

```
[Column("NameOfPerson", TypeName = "varchar", Order = 1)]
public string PersonName { get; set; }
```

Attribut [Index]

```
public class Person
{
    public int PersonID { get; set; }
    public string PersonName { get; set; }

    [Index]
    public int Age { get; set; }
}
```

Crée un index de base de données pour une colonne ou un ensemble de colonnes.

```
[Index("IX_Person_Age")]
public int Age { get; set; }
```

Cela crée un index avec un nom spécifique.

```
[Index(IsUnique = true)]
public int Age { get; set; }
```

Cela crée un index unique.

```
[Index("IX_Person_NameAndAge", 1)]
public int Age { get; set; }

[Index("IX_Person_NameAndAge", 2)]
public string PersonName { get; set; }
```

Cela crée un index composite utilisant 2 colonnes. Pour ce faire, vous devez spécifier le même nom d'index et fournir un ordre de colonne.

Remarque : L'attribut Index a été introduit dans Entity Framework 6.1. Si vous utilisez une version antérieure, les informations de cette section ne s’appliquent pas.

Attribut [ForeignKey (string)]

Spécifie le nom de la clé étrangère personnalisée si une clé étrangère ne respectant pas la

convention EF est souhaitée.

```
public class Person
{
    public int IdAddress { get; set; }

    [ForeignKey(nameof(IdAddress))]
    public virtual Address HomeAddress { get; set; }
}
```

Cela peut également être utilisé lorsque vous avez plusieurs relations avec le même type d'entité.

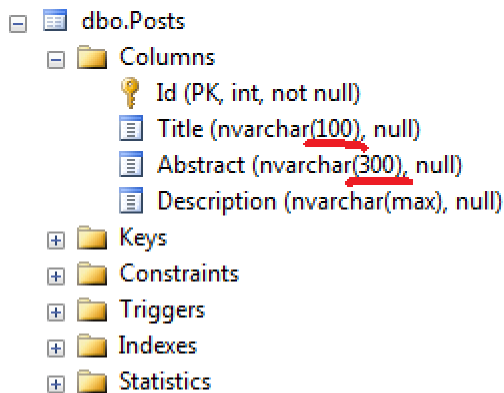
```
using System.ComponentModel.DataAnnotations.Schema;

public class Customer
{
    ...

    public int MailingAddressID { get; set; }
    public int BillingAddressID { get; set; }

    [ForeignKey("MailingAddressID")]
    public virtual Address MailingAddress { get; set; }
    [ForeignKey("BillingAddressID")]
    public virtual Address BillingAddress { get; set; }
}
```

Sans les attributs `ForeignKey` , EF pourrait les mélanger et utiliser la valeur de `BillingAddressID` lors de l'extraction de `MailingAddress` , ou il pourrait simplement proposer un nom différent pour la colonne en fonction de ses propres conventions de nommage (comme `Address_MailingAddress_Id`) et essayer de l'utiliser. au lieu de cela (ce qui entraînerait une erreur si vous l'utilisez avec une base de données existante).

[StringLength (int)] attribut

```
using System.ComponentModel.DataAnnotations;

public class Post
{
    public int Id { get; set; }

    [StringLength(100)]
    public string Title { get; set;}

    [StringLength(300)]
    public string Abstract { get; set; }

    public string Description { get; set; }
}
```

Définit une longueur maximale pour un champ de chaîne.

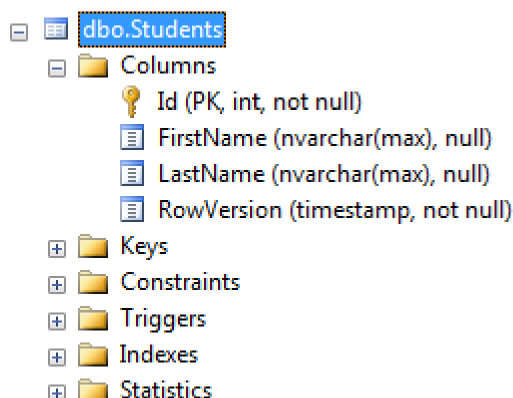


Remarque : il peut également être utilisé avec asp.net-mvc comme attribut de validation.

Attribut [Horodatage]

L' attribut **[TimeStamp]** ne peut être appliqué qu'à une propriété de tableau d'octets dans une classe d'entité donnée. Entity Framework créera une colonne d'horodatage non nullable dans la table de base de données pour cette propriété. Entity Framework utilisera automatiquement cette colonne TimeStamp dans la vérification de la concurrence.

```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    public int Id { set; get; }

    public string FirstName { set; get; }

    public string LastName { set; get; }

    [Timestamp]
    public byte[] RowVersion { get; set; }
}
```



[ConcurrencyCheck] Attribut

Cet attribut est appliqué à la propriété de classe. Vous pouvez utiliser l'attribut ConcurrencyCheck lorsque vous souhaitez utiliser des colonnes existantes pour la vérification de la concurrence et non une colonne d'horodatage distincte pour la concurrence.

```
using System.ComponentModel.DataAnnotations;

public class Author
{
    public int AuthorId { get; set; }

    [ConcurrencyCheck]
    public string AuthorName { get; set; }
}
```

Par exemple, l'attribut ConcurrencyCheck est appliqué à la propriété AuthorName de la classe Author. Ainsi, Code-First inclura la colonne AuthorName dans la commande update (clause where) pour vérifier la concurrence optimiste.

[InverseProperty (string)] attribut

```
using System.ComponentModel.DataAnnotations.Schema;

public class Department
{
    ...

    public virtual ICollection<Employee> PrimaryEmployees { get; set; }
    public virtual ICollection<Employee> SecondaryEmployees { get; set; }
}

public class Employee
{
    ...

    [InverseProperty("PrimaryEmployees")]
    public virtual Department PrimaryDepartment { get; set; }

    [InverseProperty("SecondaryEmployees")]
    public virtual Department SecondaryDepartment { get; set; }
}
```

InverseProperty peut être utilisé pour identifier des relations *bidirectionnelles* lorsque **plusieurs** relations *bidirectionnelles* existent entre deux entités.

Il indique à Entity Framework les propriétés de navigation auxquelles il doit correspondre avec les propriétés de l'autre côté.

Entity Framework ne sait pas quelle carte de propriété de navigation avec quelles propriétés de l'autre côté lorsque plusieurs relations bidirectionnelles existent entre deux entités.

Il a besoin du nom de la propriété de navigation correspondante dans la classe associée en tant que paramètre.

Cela peut également être utilisé pour les entités qui ont une relation avec d'autres entités du même type, formant une relation récursive.

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
```

```
public class TreeNode
{
    [Key]
    public int ID { get; set; }
    public int ParentID { get; set; }

    ...

    [ForeignKey("ParentID")]
    public TreeNode ParentNode { get; set; }
    [InverseProperty("ParentNode")]
    public virtual ICollection<TreeNode> ChildNodes { get; set; }
}
```

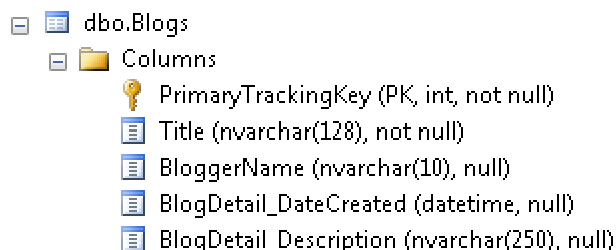
Notez également l'utilisation de l'attribut `ForeignKey` pour spécifier la colonne utilisée pour la clé étrangère sur la table. Dans le premier exemple, l'attribut `ForeignKey` la classe `Employee` pouvait être utilisé pour définir les noms de colonne.

Attribut [ComplexType]

```
using System.ComponentModel.DataAnnotations.Schema;

[ComplexType]
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

    [MaxLength(250)]
    public string Description { get; set; }
}

public class Blog
{
    ...

    public BlogDetails BlogDetail { get; set; }
}
```

Marquez la classe en tant que type complexe dans Entity Framework.

Les types complexes (ou *objets de valeur* dans la conception pilotée par un domaine) ne peuvent pas être suivis par eux-mêmes, mais ils font l'objet d'un suivi dans le cadre d'une entité. C'est pourquoi BlogDetails dans l'exemple n'a pas de propriété clé.



Ils peuvent être utiles pour décrire des entités de domaine dans plusieurs classes et superposer ces classes dans une entité complète.

Lire Code First DataAnnotations en ligne: https://riptutorial.com/fr/entity-framework/topic/4161/code-first-dataannotations

Chapitre 8: Conventions de code premier

Remarques

Convention est un ensemble de règles par défaut permettant de configurer automatiquement un modèle conceptuel basé sur les définitions de classe de domaine lorsque vous travaillez avec Code-First. Les conventions Code-First sont définies dans l'espace de noms *System.Data.Entity.ModelConfiguration.Conventions* (EF 5 & EF 6).

Examples

Convention clé primaire

Par défaut, une propriété est une clé primaire si une propriété d'une classe s'appelle «ID» (non sensible à la casse), ou le nom de la classe suivi de «ID». Si le type de la propriété de clé primaire est numérique ou GUID, il sera configuré en tant que colonne d'identité. Exemple simple:

```
public class Room
{
    // Primary key
    public int RoomId{ get; set; }
    ...
}
```

Suppression de conventions

Vous pouvez supprimer l'une des conventions définies dans l'espace de noms System.Data.Entity.ModelConfiguration.Conventions, en `OnModelCreating` méthode `OnModelCreating` .

L'exemple suivant supprime PluralizingTableNameConvention.

```
public class EshopContext : DbContext
{
    public DbSet<Product> Products { set; get; }
    . . .

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

Par défaut, EF créera une table de base de données avec le nom de la classe d'entité suffixé par «s». Dans cet exemple, le code est d' abord configuré pour ignorer la convention PluralizingTableName donc, au lieu de `dbo.Products` tableau `dbo.Product` table sera créée.

Type de découverte

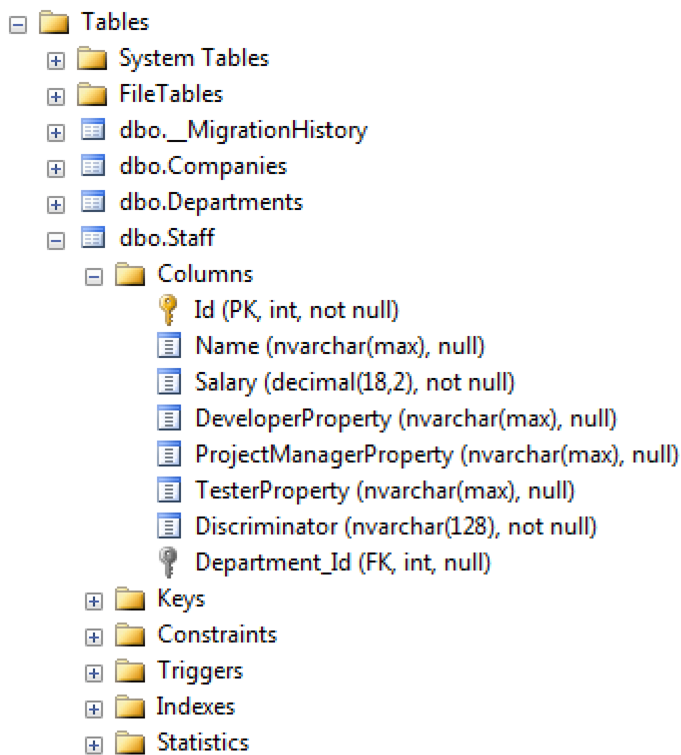
Par défaut, le code d'abord inclut dans le modèle

1. Types définis en tant que propriété DbSet dans la classe de contexte.
2. Les types de référence inclus dans les types d'entités même s'ils sont définis dans des assemblages différents.
3. Classes dérivées même si seule la classe de base est définie comme propriété DbSet

Voici un exemple, nous ajoutons seulement `Company` comme `DbSet<Company>` dans notre classe de contexte:

```
public class Company
{
    public int Id { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Department> Departments { set; get; }
}

public class Department
{
    public int Id { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Person> Staff { set; get; }
}

[Table("Staff")]
public class Person
{
    public int Id { set; get; }
    public string Name { set; get; }
    public decimal Salary { set; get; }
}

public class ProjectManager : Person
{
   public string ProjectManagerProperty { set; get; }
}

public class Developer : Person
{
    public string DeveloperProperty { set; get; }
}

public class Tester : Person
{
    public string TesterProperty { set; get; }
}

public class ApplicationDbContext : DbContext
{
    public DbSet<Company> Companies { set; get; }
}
```

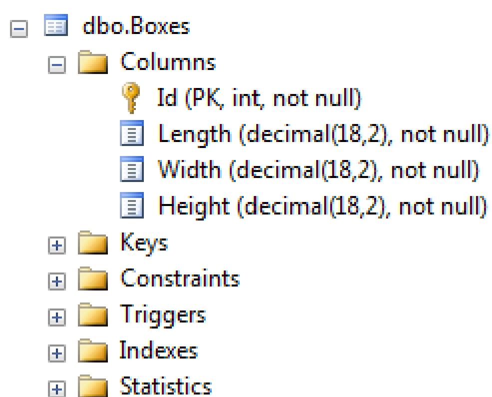
Nous pouvons voir que toutes les classes sont incluses dans le modèle



DecimalPropertyConvention

Par défaut, Entity Framework mappe les propriétés décimales sur des colonnes décimales (18,2) dans les tables de base de données.

```
public class Box
{
    public int Id { set; get; }
    public decimal Length { set; get; }
    public decimal Width { set; get; }
    public decimal Height { set; get; }
}
```

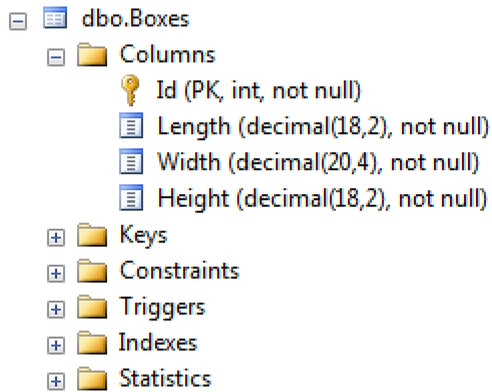


Nous pouvons changer la précision des propriétés décimales:

1.Utilisez l'API Fluent:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Box>().Property(b => b.Width).HasPrecision(20, 4);
```

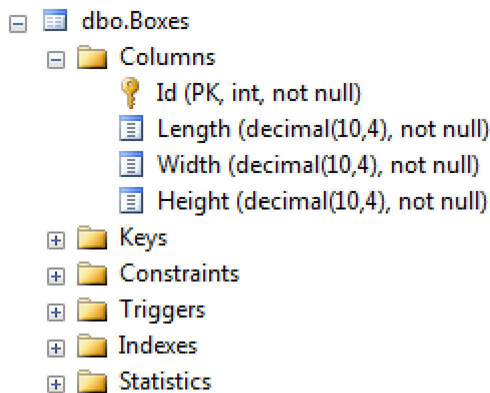
```
}
```



Seule la propriété "Width" est mappée en décimal (20, 4).

2. Remplacez la convention:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<DecimalPropertyConvention>();
    modelBuilder.Conventions.Add(new DecimalPropertyConvention(10, 4));
}
```



Chaque propriété décimale est mappée sur des colonnes décimales (10,4).

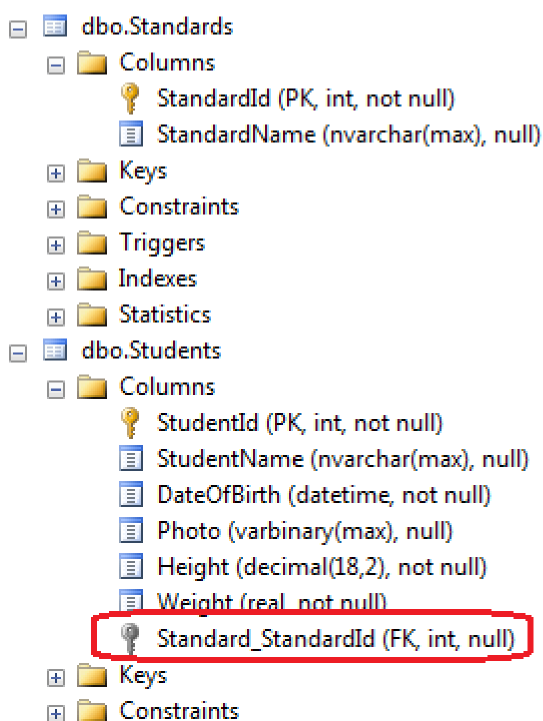
Convention Relationnelle

Code Commencez par déduire la relation entre les deux entités en utilisant la propriété de navigation. Cette propriété de navigation peut être un type de référence simple ou un type de collection. Par exemple, nous avons défini la propriété de navigation Standard dans la propriété Student et la propriété de navigation ICollection dans la classe Standard. Ainsi, Code First a automatiquement créé une relation un-à-plusieurs entre Standards et la table des étudiants en insérant la colonne de clé étrangère Standard_StandardId dans la table Students.

```
public class Student
{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
```

```
    public DateTime DateOfBirth { get; set; }

    //Navigation property
    public Standard Standard { get; set; }
}

public class Standard
{

    public int StandardId { get; set; }
    public string StandardName { get; set; }

    //Collection navigation property
    public IList<Student> Students { get; set; }

}
```

Les entités ci-dessus ont créé la relation suivante à l'aide de la clé étrangère Standard_StandardId.



Convention sur la clé étrangère

Si la classe A est en relation avec la classe B et que la classe B a une propriété avec le même nom et le même type que la clé primaire de A, alors EF suppose automatiquement que la propriété est une clé étrangère.

```
public class Department
{
    public int DepartmentId { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Person> Staff { set; get; }
}
```

```
public class Person
{
    public int Id { set; get; }
    public string Name { set; get; }
    public decimal Salary { set; get; }
    public int DepartmentId { set; get; }
    public virtual Department Department { set; get; }
}
```

Dans ce cas, DepartmentId est une clé étrangère sans spécification explicite.

Lire Conventions de code premier en ligne: https://riptutorial.com/fr/entity-framework/topic/2447/conventions-de-code-premier

Chapitre 9: Entity Framework avec SQLite

Introduction

SQLite est une base de données SQL transactionnelle autonome et sans serveur. Il peut être utilisé dans une application .NET en utilisant à la fois une bibliothèque .NET SQLite disponible gratuitement et un fournisseur Entity Framework SQLite. Cette rubrique entrera dans la configuration et l'utilisation du fournisseur Entity Framework SQLite.

Examples

Configuration d'un projet pour utiliser Entity Framework avec un fournisseur SQLite

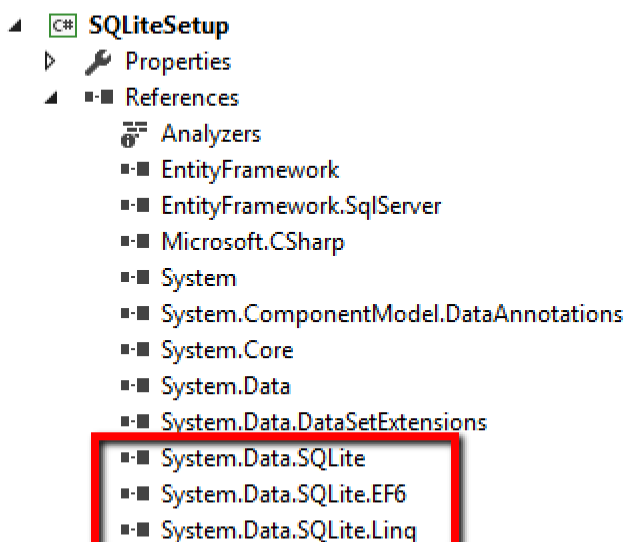
La bibliothèque Entity Framework est fournie uniquement avec un fournisseur SQL Server. Utiliser SQLite nécessitera des dépendances et une configuration supplémentaires. Toutes les dépendances requises sont disponibles sur NuGet.

Installer les bibliothèques gérées SQLite

Toutes les dépendances managées peuvent être installées à l'aide de la console NuGet Package Manager. Exécutez la commande `Install-Package System.Data.SQLite` .

```
PM> Install-Package System.Data.SQLite
Attempting to gather dependency information for package 'System.Data.SQLite.1.0.104' with respect to proje
Attempting to resolve dependencies for package 'System.Data.SQLite.1.0.104' with DependencyBehavior 'Lowes
Resolving actions to install package 'System.Data.SQLite.1.0.104'
Resolved actions to install package 'System.Data.SQLite.1.0.104'
Adding package 'EntityFramework.6.0.0' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Projec
Added package 'EntityFramework.6.0.0' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Project
Added package 'EntityFramework.6.0.0' to 'packages.config'
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Ent
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Ent

Type 'get-help EntityFramework' to see all available Entity Framework commands.
Successfully installed 'EntityFramework 6.0.0' to SQLiteSetup
Adding package 'System.Data.SQLite.Core.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2
Added package 'System.Data.SQLite.Core.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 20
Added package 'System.Data.SQLite.Core.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite.Core 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.EF6.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 20
Added package 'System.Data.SQLite.EF6.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 201
Added package 'System.Data.SQLite.EF6.1.0.104' to 'packages.config'
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Sys
Successfully installed 'System.Data.SQLite.EF6 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.Linq.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2
Added package 'System.Data.SQLite.Linq.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 20
Added package 'System.Data.SQLite.Linq.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite.Linq 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.1.0.104', which only has dependencies, to project 'SQLiteSetup'.
Adding package 'System.Data.SQLite.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\P
Added package 'System.Data.SQLite.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Pr
Added package 'System.Data.SQLite.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite 1.0.104' to SQLiteSetup
```

Comme indiqué ci-dessus, lors de l'installation de `System.Data.SQLite` , toutes les bibliothèques gérées associées sont installées avec elle. Cela inclut `System.Data.SQLite.EF6` , le fournisseur EF pour SQLite. Le projet fait également maintenant référence aux assemblys requis pour utiliser le fournisseur SQLite.



Y compris la bibliothèque non gérée

Les bibliothèques gérées SQLite dépendent d'un assembly non géré nommé `SQLite.Interop.dll` . Il est inclus dans les assemblys de packages téléchargés avec le package SQLite et ils sont

automatiquement copiés dans votre répertoire de génération lorsque vous générez le projet. Cependant, comme il n'est pas géré, il ne sera pas inclus dans votre liste de références. Mais notez que cet assemblage est le plus distribué avec l'application pour que les assemblys SQLite fonctionnent.

Remarque: Cet assemblage dépend des bits, ce qui signifie que vous devrez inclure un assemblage spécifique pour chaque bitness que vous prévoyez de prendre en charge (x86 / x64).

Modifier le fichier App.config du projet

Le fichier `app.config` nécessitera certaines modifications avant que SQLite puisse être utilisé en tant que fournisseur Entity Framework.

Correctifs requis

Lors de l'installation du package, le fichier `app.config` est automatiquement mis à jour pour inclure les entrées nécessaires pour SQLite et SQLite EF. Malheureusement, ces entrées contiennent des erreurs. Ils doivent être modifiés avant de fonctionner correctement.

Tout d'abord, `DbProviderFactories` élément `DbProviderFactories` dans le fichier de configuration. Il se trouve dans l'élément `system.data` et contiendra les éléments suivants

```
<DbProviderFactories>
  <remove invariant="System.Data.SQLite.EF6" />
  <add name="SQLite Data Provider (Entity Framework 6)" invariant="System.Data.SQLite.EF6"
description=".NET Framework Data Provider for SQLite (Entity Framework 6)"
type="System.Data.SQLite.EF6.SQLiteProviderFactory, System.Data.SQLite.EF6" />
  <remove invariant="System.Data.SQLite" /><add name="SQLite Data Provider"
invariant="System.Data.SQLite" description=".NET Framework Data Provider for SQLite"
type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite" />
</DbProviderFactories>
```

Cela peut être simplifié pour contenir une seule entrée

```
<DbProviderFactories>
    <add name="SQLite Data Provider" invariant="System.Data.SQLite.EF6" description=".NET
Framework Data Provider for SQLite" type="System.Data.SQLite.SQLiteFactory,
System.Data.SQLite" />
</DbProviderFactories>
```

Avec cela, nous avons spécifié que les fournisseurs EF6 SQLite devraient utiliser l'usine SQLite.

Ajouter une chaîne de connexion SQLite

Les chaînes de connexion peuvent être ajoutées au fichier de configuration dans l'élément racine. Ajoutez une chaîne de connexion pour accéder à une base de données SQLite.

```
<connectionStrings>
    <add name="TestContext" connectionString="data source=testdb.sqlite;initial
catalog=Test;App=EntityFramework;" providerName="System.Data.SQLite.EF6"/>
</connectionStrings>
```

La chose importante à noter ici est le `provider` . Il a été défini sur `System.Data.SQLite.EF6` . Cela indique à EF que lorsque nous utilisons cette chaîne de connexion, nous voulons utiliser SQLite. La `data source` spécifiée n'est qu'un exemple et dépendra de l'emplacement et du nom de votre base de données SQLite.

Votre premier SQLite DbContext

Une fois l'installation et la configuration terminées, vous pouvez maintenant commencer à utiliser `DbContext` qui fonctionnera sur votre base de données SQLite.

```
public class TestContext : DbContext
{
    public TestContext()
        : base("name=TestContext") { }
}
```

En spécifiant `name=TestContext` , j'ai indiqué que la chaîne de connexion TestContext située dans le fichier `app.config` doit être utilisée pour créer le contexte. Cette chaîne de connexion a été configurée pour utiliser SQLite, ce contexte utilisera donc une base de données SQLite.

Lire Entity Framework avec SQLite en ligne: https://riptutorial.com/fr/entity-framework/topic/9280/entity-framework-avec-sqlite

Chapitre 10: Entity-Framework avec Postgresql

Examples

Pré-étapes nécessaires pour utiliser Entity Framework 6.1.3 avec PostgresSql en utilisant Npgsqlddexprovider

1) Sauvegarde de Machine.config à partir des emplacements C: \ Windows \ Microsoft.NET \ Framework \ v4.0.30319 \ Config et C: \ Windows \ Microsoft.NET \ Framework64 \ v4.0.30319 \ Config

2) Copiez-les à un autre endroit et modifiez-les en tant que

a) localisez et ajoutez sous `<system.data> <DbProviderFactories>`

```
            <add name="Npgsql Data Provider" invariant="Npgsql" support="FF"
            description=".Net Framework Data Provider for Postgresql Server"
            type="Npgsql.NpgsqlFactory, Npgsql, Version=2.2.5.0, Culture=neutral,
PublicKeyToken=5d8b90d52f46fda7" />
```

b) s'il existe déjà au-dessus de l'entrée, vérifiez la version et mettez-la à jour.

3. Remplacez les fichiers originaux par des fichiers modifiés.
4. exécutez l'invite de commande du développeur pour VS2013 en tant qu'administrateur.
5. Si Npgsql est déjà installé, utilisez la commande "gacutil -u Npgsql" pour désinstaller puis installez la nouvelle version de Npgsql 2.5.0 par la commande "gacutil -i [chemin de dll]"
6. Faites ci-dessus pour Mono.Security 4.0.0.0
7. Téléchargez NpgsqlDdexProvider-2.2.0-VS2013.zip et exécutez NpgsqlDdexProvider.vsix à partir de celui-ci (fermez toutes les instances de Visual Studio)
8. EFTools6.1.3-beta1ForVS2013.msi trouvé et lancez-le.
9. Après avoir créé le nouveau projet, installez la version d'EntityFramework (6.1.3), NpgSql (2.5.0) et NpgSql.EntityFramework (2.5.0) à partir de Manage Nuget Packages.10) Sa fin est atteinte ... Ajouter un nouveau modèle de données d'entité dans votre projet MVc

Lire Entity-Framework avec Postgresql en ligne: https://riptutorial.com/fr/entity-framework/topic/7647/entity-framework-avec-postgresql

Chapitre 11: État de l'entité de gestion

Remarques

Les entités dans Entity Framework peuvent avoir différents états répertoriés par l'énumération System.Data.Entity.EntityState . Ces états sont:

```
Added
Deleted
Detached
Modified
Unchanged
```

Entity Framework fonctionne avec les POCO. Cela signifie que les entités sont des classes simples qui n'ont pas de propriétés et méthodes pour gérer leur propre état. L'état de l'entité est géré par un contexte lui-même, dans `ObjectStateManager` .

Cette rubrique couvre différentes manières de définir l'état d'une entité.

Examples

Définition de l'état Ajout d'une entité unique

`EntityState.Added` peut être configuré de deux manières totalement équivalentes:

1. En définissant l'état de son entrée dans le contexte:

   ```
   context.Entry(entity).State = EntityState.Added;
   ```

2. En l'ajoutant à un `DbSet` du contexte:

   ```
   context.Entities.Add(entity);
   ```

Lors de l'appel de `SaveChanges` , l'entité sera insérée dans la base de données. Lorsqu'elle a une colonne d'identité (une clé primaire à auto-incrémentation automatique), puis après `SaveChanges` , la propriété de clé primaire de l'entité contiendra la nouvelle valeur générée, *même si cette propriété avait déjà une valeur* .

Définition de l'état Ajout d'un graphique d'objet

Définir l’état d’un *graphe d’objets* (une collection d’entités liées) à `Added` est différent de définir une seule entité comme étant `Added` (voir cet exemple).

Dans l'exemple, nous stockons des planètes et leurs lunes:

Modèle de classe

```
public class Planet
{
    public Planet()
    {
        Moons = new HashSet<Moon>();
    }
    public int ID { get; set; }
    public string Name { get; set; }
    public ICollection<Moon> Moons { get; set; }
}

public class Moon
{
    public int ID { get; set; }
    public int PlanetID { get; set; }
    public string Name { get; set; }
}
```

Le contexte

```
public class PlanetDb : DbContext
{
    public property DbSet<Planet> Planets { get; set; }
}
```

Nous utilisons une instance de ce contexte pour ajouter des planètes et leurs lunes:

Exemple

```
var mars = new Planet { Name = "Mars" };
mars.Moons.Add(new Moon { Name = "Phobos" });
mars.Moons.Add(new Moon { Name = "Deimos" });

context.Planets.Add(mars);

Console.WriteLine(context.Entry(mars).State);
Console.WriteLine(context.Entry(mars.Moons.First()).State);
```

Sortie:

```
Added
Added
```

Ce que nous voyons ici, c'est que l'ajout d'une `Planet` définit également l'état d'une lune à `Added` .

Lors de la définition de l'état d'une entité à `Added` , toutes les entités de ses propriétés de navigation (propriétés qui «naviguent» vers d'autres entités, telles que `Planet.Moons`) sont également marquées comme `Added` , *sauf si elles sont déjà associées au contexte* .

Lire État de l'entité de gestion en ligne: https://riptutorial.com/fr/entity-framework/topic/5256/etat-de-l-entite-de-gestion

Chapitre 12: Héritage avec EntityFramework (Code First)

Examples

Tableau par hiérarchie

Cette approche va générer une table sur la base de données pour représenter toute la structure d'héritage.

Exemple:

```
public abstract class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
}

public class Employee : Person
{
    public DateTime AdmissionDate { get; set; }
    public string JobDescription { get; set; }
}

public class Customer : Person
{
    public DateTime LastPurchaseDate { get; set; }
    public int TotalVisits { get; set; }
}

// On DbContext
public DbSet<Person> People { get; set; }
public DbSet<Employee> Employees { get; set; }
public DbSet<Customer> Customers { get; set; }
```

La table générée sera:

Tableau: Champs de personnes: Nom de l'identifiant BirthDate Discrimitator AdmissionDate JobDescription LastPurchaseDate TotalVisits

Où 'Discriminator' contiendra le nom de la sous-classe sur l'héritage et 'AdmissionDate', 'JobDescription', 'LastPurchaseDate', 'TotalVisits' sont nullables.

Avantages

- De meilleures performances, car aucune jointure n'est requise, mais pour de nombreuses colonnes, la base de données peut nécessiter de nombreuses opérations de pagination.
- Simple à utiliser et à créer
- Facile d'ajouter plus de sous-classes et de champs

Désavantages

- Viole la 3ème forme normale Wikipedia: troisième forme normale
- Crée beaucoup de champs nullables

Tableau par type

Cette approche génère des tables (n + 1) sur la base de données pour représenter toute la structure d'héritage où n est le nombre de sous-classes.

Comment:

```
public abstract class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
}

[Table("Employees")]
public class Employee : Person
{
    public DateTime AdmissionDate { get; set; }
    public string JobDescription { get; set; }
}

[Table("Customers")]
public class Customer : Person
{
    public DateTime LastPurchaseDate { get; set; }
    public int TotalVisits { get; set; }
}

// On DbContext
public DbSet<Person> People { get; set; }
public DbSet<Employee> Employees { get; set; }
public DbSet<Customer> Customers { get; set; }
```

La table générée sera:

Tableau: Champs de personnes: Nom de l'identifiant Date de naissance

Tableau: Champs Employees: PersonId AdmissionDate JobDescription

Table: Customers: Champs: PersonId LastPurchaseDate TotalVisits

Où 'PersonId' sur toutes les tables sera une clé primaire et une contrainte pour People.Id

Avantages

- Tables normalisées
- Facile à ajouter des colonnes et des sous-classes
- Aucune colonne nullable

Désavantages

- Join est nécessaire pour récupérer les données
- L'inférence de sous-classe est plus chère

Lire Héritage avec EntityFramework (Code First) en ligne: https://riptutorial.com/fr/entity-framework/topic/7715/heritage-avec-entityframework--code-first-

Chapitre 13: Initialiseurs de base de données

Examples

CreateDatabaseIfNotExists

Implémentation de `IDatabaseInitializer` utilisée par défaut dans EntityFramework. Comme son nom l'indique, il crée la base de données si aucune n'existe. Cependant, lorsque vous modifiez le modèle, il génère une exception.

Usage:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new CreateDatabaseIfNotExists<MyContext>());
    }
}
```

DropCreateDatabaseIfModelChanges

Cette implémentation de `IDatabaseInitializer` supprime et recrée la base de données si le modèle change automatiquement.

Usage:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<MyContext>());
    }
}
```

DropCreateDatabaseAlways

Cette implémentation de `IDatabaseInitializer` supprime et recrée la base de données chaque fois que votre contexte est utilisé dans le domaine d'application des applications. Méfiez-vous de la perte de données due au fait que la base de données est recréée.

Usage:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new DropCreateDatabaseAlways<MyContext>());
    }
}
```

Initialiseur de base de données personnalisé

Vous pouvez créer votre propre implémentation d' `IDatabaseInitializer` .

Exemple d'implémentation d'un initialiseur, qui va migrer la base de données vers 0, puis migrer jusqu'à la migration la plus récente (utile, par exemple, lors de l'exécution de tests d'intégration). Pour ce faire, vous avez également besoin d'un type `DbMigrationsConfiguration` .

```
public class RecreateFromScratch<TContext, TMigrationsConfiguration> :
IDatabaseInitializer<TContext>
where TContext : DbContext
where TMigrationsConfiguration : DbMigrationsConfiguration<TContext>, new()
{
    private readonly DbMigrationsConfiguration<TContext> _configuration;

    public RecreateFromScratch()
    {
        _configuration = new TMigrationsConfiguration();
    }

    public void InitializeDatabase(TContext context)
    {
        var migrator = new DbMigrator(_configuration);
        migrator.Update("0");
        migrator.Update();
    }
}
```

MigrateDatabaseToLatestVersion

Une implémentation de `IDatabaseInitializer` qui utilisera Code First Migrations pour mettre à jour la base de données vers la dernière version. Pour utiliser cet initialiseur, vous devez également utiliser le type `DbMigrationsConfiguration` .

Usage:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(
            new MigrateDatabaseToLatestVersion<MyContext, Configuration>());
    }
}
```

Lire Initialiseurs de base de données en ligne: https://riptutorial.com/fr/entity-framework/topic/5526/initialiseurs-de-base-de-donnees

Chapitre 14: Meilleures pratiques pour l'entité Framework (Simple & Professional)

Introduction

Cet article présente une pratique simple et professionnelle pour utiliser Entity Framework.

Simple: car il ne nécessite qu'une classe (avec une interface)

Professionnel: parce qu'il applique les principes de l'architecture SOLID

Je ne veux pas parler plus… profitons-en!

Examples

1- Entity Framework @ Data layer (Bases)

Dans cet article, nous utiliserons une base de données simple appelée «Company» avec deux tables:

[dbo]. [Catégories] ([CategoryID], [CategoryName])

[dbo]. [Produits] ([ProductID], [CategoryID], [ProductName])

1-1 Générer le code de l'entité d'entité

Dans cette couche, nous générons le code Entity Framework (dans la bibliothèque du projet) (voir cet article pour savoir comment faire), puis vous aurez les classes suivantes:

```
public partial class CompanyContext : DbContext
public partial class Product
public partial class Category
```

1-2 Créer une interface de base

Nous allons créer une interface pour nos fonctions de base

```
public interface IDbRepository : IDisposable
{
    #region Tables and Views functions

    IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class;
    TEntity Add<TEntity>(TEntity entity) where TEntity : class;
    TEntity Delete<TEntity>(TEntity entity) where TEntity : class;
    TEntity Attach<TEntity>(TEntity entity) where TEntity : class;
    TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class;

    #endregion Tables and Views functions
```

```
    #region Transactions Functions

    int Commit();
    Task<int> CommitAsync(CancellationToken cancellationToken = default(CancellationToken));

    #endregion Transactions Functions

    #region Database Procedures and Functions

    TResult Execute<TResult>(string functionName, params object[] parameters);

    #endregion Database Procedures and Functions
}
```

1-3 Mise en œuvre de l'interface de base

```
/// <summary>
/// Implementing basic tables, views, procedures, functions, and transaction functions
/// Select (GetAll), Insert (Add), Delete, and Attach
/// No Edit (Modify) function (can modify attached entity without function call)
/// Executes database procedures or functions (Execute)
/// Transaction functions (Commit)
/// More functions can be added if needed
/// </summary>
/// <typeparam name="TEntity">Entity Framework table or view</typeparam>
public class DbRepository : IDbRepository
{
    #region Protected Members

    protected DbContext _dbContext;

    #endregion Protected Members

    #region Constractors

    /// <summary>
    /// Repository constructor
    /// </summary>
    /// <param name="dbContext">Entity framework databse context</param>
    public DbRepository(DbContext dbContext)
    {
        _dbContext = dbContext;

        ConfigureContext();
    }

    #endregion Constractors

    #region IRepository Implementation

    #region Tables and Views functions

    /// <summary>
    /// Query all
    /// Set noTracking to true for selecting only (read-only queries)
    /// Set noTracking to false for insert, update, or delete after select
    /// </summary>
    public virtual IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult :
class
```

```
{
    var entityDbSet = GetDbSet<TResult>();

    if (noTracking)
        return entityDbSet.AsNoTracking();

    return entityDbSet;
}

public virtual TEntity Add<TEntity>(TEntity entity) where TEntity : class
{
    return GetDbSet<TEntity>().Add(entity);
}

/// <summary>
/// Delete loaded (attached) or unloaded (Detached) entitiy
/// No need to load object to delete it
/// Create new object of TEntity and set the id then call Delete function
/// </summary>
/// <param name="entity">TEntity</param>
/// <returns></returns>
public virtual TEntity Delete<TEntity>(TEntity entity) where TEntity : class
{
    if (_dbContext.Entry(entity).State == EntityState.Detached)
    {
        _dbContext.Entry(entity).State = EntityState.Deleted;
        return entity;
    }
    else
        return GetDbSet<TEntity>().Remove(entity);
}

public virtual TEntity Attach<TEntity>(TEntity entity) where TEntity : class
{
    return GetDbSet<TEntity>().Attach(entity);
}

public virtual TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class
{
    if (_dbContext.Entry(entity).State == EntityState.Detached)
        return Attach(entity);

    return entity;
}

#endregion Tables and Views functions

#region Transactions Functions

/// <summary>
/// Saves all changes made in this context to the underlying database.
/// </summary>
/// <returns>The number of objects written to the underlying database.</returns>
public virtual int Commit()
{
    return _dbContext.SaveChanges();
}

/// <summary>
/// Asynchronously saves all changes made in this context to the underlying database.
/// </summary>
```

```
    /// <param name="cancellationToken">A System.Threading.CancellationToken to observe while
waiting for the task to complete.</param>
    /// <returns>A task that represents the asynchronous save operation.  The task result
contains the number of objects written to the underlying database.</returns>
    public virtual Task<int> CommitAsync(CancellationToken cancellationToken =
default(CancellationToken))
    {
        return _dbContext.SaveChangesAsync(cancellationToken);
    }

    #endregion Transactions Functions

    #region Database Procedures and Functions

    /// <summary>
    /// Executes any function in the context
    /// use to call database procesdures and functions
    /// </summary>>
    /// <typeparam name="TResult">return function type</typeparam>
    /// <param name="functionName">context function name</param>
    /// <param name="parameters">context function parameters in same order</param>
    public virtual TResult Execute<TResult>(string functionName, params object[] parameters)
    {
        MethodInfo method = _dbContext.GetType().GetMethod(functionName);

        return (TResult)method.Invoke(_dbContext, parameters);
    }

    #endregion Database Procedures and Functions

    #endregion IRepository Implementation

    #region IDisposable Implementation

    public void Dispose()
    {
        _dbContext.Dispose();
    }

    #endregion IDisposable Implementation

    #region Protected Functions

    /// <summary>
    /// Set Context Configuration
    /// </summary>
    protected virtual void ConfigureContext()
    {
        // set your recommended Context Configuration
        _dbContext.Configuration.LazyLoadingEnabled = false;
    }

    #endregion Protected Functions

    #region Private Functions

    private DbSet<TEntity> GetDbSet<TEntity>() where TEntity : class
    {
        return _dbContext.Set<TEntity>();
    }
```

```
    #endregion Private Functions

}
```

2- Couche Entity Framework @ Business

Dans cette couche, nous écrirons l'activité applicative.

Il est recommandé pour chaque écran de présentation, vous créez l'interface métier et la classe d'implémentation qui contiennent toutes les fonctions requises pour l'écran.

Ci-dessous, nous allons écrire le business pour l'écran du produit comme exemple

```
/// <summary>
/// Contains Product Business functions
/// </summary>
public interface IProductBusiness
{
    Product SelectById(int productId, bool noTracking = true);
    Task<IEnumerable<dynamic>> SelectByCategoryAsync(int CategoryId);
    Task<Product> InsertAsync(string productName, int categoryId);
    Product InsertForNewCategory(string productName, string categoryName);
    Product Update(int productId, string productName, int categoryId);
    Product Update2(int productId, string productName, int categoryId);
    int DeleteWithoutLoad(int productId);
    int DeleteLoadedProduct(Product product);
    IEnumerable<GetProductsCategory_Result> GetProductsCategory(int categoryId);
}



/// <summary>
/// Implementing Product Business functions
/// </summary>
public class ProductBusiness : IProductBusiness
{
    #region Private Members

    private IDbRepository _dbRepository;

    #endregion Private Members

    #region Constructors

    /// <summary>
    /// Product Business Constructor
    /// </summary>
    /// <param name="dbRepository"></param>
    public ProductBusiness(IDbRepository dbRepository)
    {
        _dbRepository = dbRepository;
    }

    #endregion Constructors

    #region IProductBusiness Function

    /// <summary>
```

```
    /// Selects Product By Id
    /// </summary>
    public Product SelectById(int productId, bool noTracking = true)
    {
        var products = _dbRepository.GetAll<Product>(noTracking);

        return products.FirstOrDefault(pro => pro.ProductID == productId);
    }

    /// <summary>
    /// Selects Products By Category Id Async
    /// To have async method, add reference to EntityFramework 6 dll or higher
    /// also you need to have the namespace "System.Data.Entity"
    /// </summary>
    /// <param name="CategoryId">CategoryId</param>
    /// <returns>Return what ever the object that you want to return</returns>
    public async Task<IEnumerable<dynamic>> SelectByCategoryAsync(int CategoryId)
    {
        var products = _dbRepository.GetAll<Product>();
        var categories = _dbRepository.GetAll<Category>();

        var result = (from pro in products
                      join cat in categories
                      on pro.CategoryID equals cat.CategoryID
                      where pro.CategoryID == CategoryId
                      select new
                      {
                          ProductId = pro.ProductID,
                          ProductName = pro.ProductName,
                          CategoryName = cat.CategoryName
                      }
                              );

        return await result.ToListAsync();
    }

    /// <summary>
    /// Insert Async new product for given category
    /// </summary>
    public async Task<Product> InsertAsync(string productName, int categoryId)
    {
        var newProduct = _dbRepository.Add(new Product() { ProductName = productName,
CategoryID = categoryId });

        await _dbRepository.CommitAsync();

        return newProduct;
    }

    /// <summary>
    /// Insert new product and new category
    /// Do many database actions in one transaction
    /// each _dbRepository.Commit(); will commit one transaction
    /// </summary>
    public Product InsertForNewCategory(string productName, string categoryName)
    {
        var newCategory = _dbRepository.Add(new Category() { CategoryName = categoryName });
        var newProduct = _dbRepository.Add(new Product() { ProductName = productName, Category
= newCategory });

        _dbRepository.Commit();
```

```
        return newProduct;
    }

    /// <summary>
    /// Update given product with tracking
    /// </summary>
    public Product Update(int productId, string productName, int categoryId)
    {
        var product = SelectById(productId,false);
        product.CategoryID = categoryId;
        product.ProductName = productName;

        _dbRepository.Commit();

        return product;
    }

    /// <summary>
    /// Update given product with no tracking and attach function
    /// </summary>
    public Product Update2(int productId, string productName, int categoryId)
    {
        var product = SelectById(productId);
        _dbRepository.Attach(product);

        product.CategoryID = categoryId;
        product.ProductName = productName;

        _dbRepository.Commit();

        return product;
    }

    /// <summary>
    /// Deletes product without loading it
    /// </summary>
    public int DeleteWithoutLoad(int productId)
    {
        _dbRepository.Delete(new Product() { ProductID = productId });

        return _dbRepository.Commit();
    }

    /// <summary>
    /// Deletes product after loading it
    /// </summary>
    public int DeleteLoadedProduct(Product product)
    {
        _dbRepository.Delete(product);

        return _dbRepository.Commit();
    }

    /// <summary>
    /// Assuming we have the following procedure in database
    /// PROCEDURE [dbo].[GetProductsCategory] @CategoryID INT, @OrderBy VARCHAR(50)
    /// </summary>
    public IEnumerable<GetProductsCategory_Result> GetProductsCategory(int categoryId)
    {
        return
```

```
_dbRepository.Execute<IEnumerable<GetProductsCategory_Result>>("GetProductsCategory",
categoryId, "ProductName DESC");
    }


    #endregion IProductBusiness Function
}
```

3- Utilisation de la couche Business @ Couche de présentation (MVC)

Dans cet exemple, nous utiliserons la couche Business dans la couche Presentation. Et nous utiliserons MVC comme exemple de couche de présentation (mais vous pouvez utiliser n'importe quelle autre couche de présentation).

Nous devons d'abord enregistrer l'IoC (nous utiliserons Unity, mais vous pouvez utiliser n'importe quel IoC), puis écrivez notre couche Présentation

3-1 Enregistrer les types d'unité dans MVC

3-1-1 Ajouter «Le bootstrapper Unity pour ASP.NET MVC» BackGet

3-1-2 Ajouter UnityWebActivator.Start (); dans le fichier Global.asax.cs (fonction Application_Start ())

3-1-3 Modifier la fonction UnityConfig.RegisterTypes comme suit

```
    public static void RegisterTypes(IUnityContainer container)
    {
        // Data Access Layer
        container.RegisterType<DbContext, CompanyContext>(new PerThreadLifetimeManager());
        container.RegisterType(typeof(IDbRepository), typeof(DbRepository), new
PerThreadLifetimeManager());

        // Business Layer
        container.RegisterType<IProductBusiness, ProductBusiness>(new
PerThreadLifetimeManager());

    }
```

3-2 Utilisation de la couche de gestion @ Couche de présentation (MVC)

```
public class ProductController : Controller
{
    #region Private Members

    IProductBusiness _productBusiness;

    #endregion Private Members

    #region Constractors

    public ProductController(IProductBusiness productBusiness)
    {
        _productBusiness = productBusiness;
    }
```

```
    #endregion Constractors

    #region Action Functions

    [HttpPost]
    public ActionResult InsertForNewCategory(string productName, string categoryName)
    {
        try
        {
            // you can use any of IProductBusiness functions
            var newProduct = _productBusiness.InsertForNewCategory(productName, categoryName);

            return Json(new { success = true, data = newProduct });
        }
        catch (Exception ex)
        {   /* log ex*/
            return Json(new { success = false, errorMessage = ex.Message});
        }
    }

    [HttpDelete]
    public ActionResult SmartDeleteWithoutLoad(int productId)
    {
        try
        {
            // deletes product without load
            var deletedProduct = _productBusiness.DeleteWithoutLoad(productId);

            return Json(new { success = true, data = deletedProduct });
        }
        catch (Exception ex)
        {   /* log ex*/
            return Json(new { success = false, errorMessage = ex.Message });
        }
    }

    public async Task<ActionResult> SelectByCategoryAsync(int CategoryId)
    {
        try
        {
            var results = await _productBusiness.SelectByCategoryAsync(CategoryId);

            return Json(new { success = true, data = results },JsonRequestBehavior.AllowGet);
        }
        catch (Exception ex)
        {   /* log ex*/
            return Json(new { success = false, errorMessage = ex.Message
},JsonRequestBehavior.AllowGet);
        }
    }
    #endregion Action Functions
}
```

4- Entity Framework @ Unit Test Layer

Dans la couche Test unitaire, nous testons généralement les fonctionnalités de la couche métier. Et pour ce faire, nous allons supprimer les dépendances Data Layer (Entity Framework).

Et la question est la suivante: comment puis-je supprimer les dépendances Entity Framework afin de tester les fonctions de la couche métier?

Et la réponse est simple: nous allons faire une fausse implémentation pour l'interface IDbRepository puis nous pouvons faire notre test unitaire

4-1 Implémentation de l'interface de base (fausse implémentation)

```
class FakeDbRepository : IDbRepository
{
    #region Protected Members

    protected Hashtable _dbContext;
    protected int _numberOfRowsAffected;
    protected Hashtable _contextFunctionsResults;

    #endregion Protected Members

    #region Constractors

    public FakeDbRepository(Hashtable contextFunctionsResults = null)
    {
        _dbContext = new Hashtable();
        _numberOfRowsAffected = 0;
        _contextFunctionsResults = contextFunctionsResults;
    }

    #endregion Constractors

    #region IRepository Implementation

    #region Tables and Views functions

    public IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class
    {
        return GetDbSet<TResult>().AsQueryable();
    }

    public TEntity Add<TEntity>(TEntity entity) where TEntity : class
    {
        GetDbSet<TEntity>().Add(entity);
        ++_numberOfRowsAffected;
        return entity;
    }

    public TEntity Delete<TEntity>(TEntity entity) where TEntity : class
    {
        GetDbSet<TEntity>().Remove(entity);
        ++_numberOfRowsAffected;
        return entity;
    }

    public TEntity Attach<TEntity>(TEntity entity) where TEntity : class
    {
        return Add(entity);
    }

    public TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class
    {
```

```
        if (!GetDbSet<TEntity>().Contains(entity))
            return Attach(entity);

        return entity;
    }

    #endregion Tables and Views functions

    #region Transactions Functions


    public virtual int Commit()
    {
        var numberOfRowsAffected = _numberOfRowsAffected;
        _numberOfRowsAffected = 0;
        return numberOfRowsAffected;
    }

    public virtual Task<int> CommitAsync(CancellationToken cancellationToken =
default(CancellationToken))
    {
        var numberOfRowsAffected = _numberOfRowsAffected;
        _numberOfRowsAffected = 0;
        return new Task<int>(() => numberOfRowsAffected);
    }

    #endregion Transactions Functions

    #region Database Procedures and Functions

    public virtual TResult Execute<TResult>(string functionName, params object[] parameters)
    {
        if (_contextFunctionsResults != null &&
_contextFunctionsResults.Contains(functionName))
            return (TResult)_contextFunctionsResults[functionName];

        throw new NotImplementedException();
    }

    #endregion Database Procedures and Functions

    #endregion IRepository Implementation

    #region IDisposable Implementation

    public void Dispose()
    {

    }

    #endregion IDisposable Implementation

    #region Private Functions

    private List<TEntity> GetDbSet<TEntity>() where TEntity : class
    {
        if (!_dbContext.Contains(typeof(TEntity)))
            _dbContext.Add(typeof(TEntity), new List<TEntity>());

        return (List<TEntity>)_dbContext[typeof(TEntity)];
    }
```

```
    #endregion Private Functions
}
```

4-2 Exécuter vos tests unitaires

```
[TestClass]
public class ProductUnitTest
{
    [TestMethod]
    public void TestInsertForNewCategory()
    {
        // Initialize repositories
        FakeDbRepository _dbRepository = new FakeDbRepository();

        // Initialize Business object
        IProductBusiness productBusiness = new ProductBusiness(_dbRepository);

        // Process test method
        productBusiness.InsertForNewCategory("Test Product", "Test Category");

        int _productCount = _dbRepository.GetAll<Product>().Count();
        int _categoryCount = _dbRepository.GetAll<Category>().Count();

        Assert.AreEqual<int>(1, _productCount);
        Assert.AreEqual<int>(1, _categoryCount);
    }

    [TestMethod]
    public void TestProceduresFunctionsCall()
    {
        // Initialize Procedures / Functions result
        Hashtable _contextFunctionsResults = new Hashtable();
        _contextFunctionsResults.Add("GetProductsCategory", new
List<GetProductsCategory_Result> {
            new GetProductsCategory_Result() { ProductName = "Product 1", ProductID = 1,
CategoryName = "Category 1" },
            new GetProductsCategory_Result() { ProductName = "Product 2", ProductID = 2,
CategoryName = "Category 1" },
            new GetProductsCategory_Result() { ProductName = "Product 3", ProductID = 3,
CategoryName = "Category 1" }});

        // Initialize repositories
        FakeDbRepository _dbRepository = new FakeDbRepository(_contextFunctionsResults);

        // Initialize Business object
        IProductBusiness productBusiness = new ProductBusiness(_dbRepository);

        // Process test method
        var results = productBusiness.GetProductsCategory(1);

        Assert.AreEqual<int>(3, results.Count());
    }
}
```

Lire Meilleures pratiques pour l'entité Framework (Simple & Professional) en ligne: https://riptutorial.com/fr/entity-framework/topic/8879/meilleures-pratiques-pour-l-entite-framework--simple--amp--professional-

Chapitre 15: Modèles de contraintes

Examples

Relations un-à-plusieurs

UserType appartient à de nombreux utilisateurs <-> Les utilisateurs ont un seul type d'utilisateur

Propriété de navigation à sens unique avec requis

```
public class UserType
{
    public int UserTypeId {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int UserTypeId {get; set;}
    public virtual UserType UserType {get; set;}
}

Entity<User>().HasRequired(u => u.UserType).WithMany().HasForeignKey(u => u.UserTypeId);
```

Propriété de navigation à sens unique avec facultatif (la clé étrangère doit être de type `Nullable`)

```
public class UserType
{
    public int UserTypeId {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int? UserTypeId {get; set;}
    public virtual UserType UserType {get; set;}
}

Entity<User>().HasOptional(u => u.UserType).WithMany().HasForeignKey(u => u.UserTypeId);
```

Propriété de navigation bidirectionnelle avec (obligatoire / facultatif modifie la propriété de la clé étrangère selon les besoins)

```
public class UserType
{
    public int UserTypeId {get; set;}
    public virtual ICollection<User> Users {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int UserTypeId {get; set;}
    public virtual UserType UserType {get; set;}
}
```

Champs obligatoires

```
Entity<User>().HasRequired(u => u.UserType).WithMany(ut => ut.Users).HasForeignKey(u =>
u.UserTypeId);
```

Optionnel

```
Entity<User>().HasOptional(u => u.UserType).WithMany(ut => ut.Users).HasForeignKey(u =>
u.UserTypeId);
```

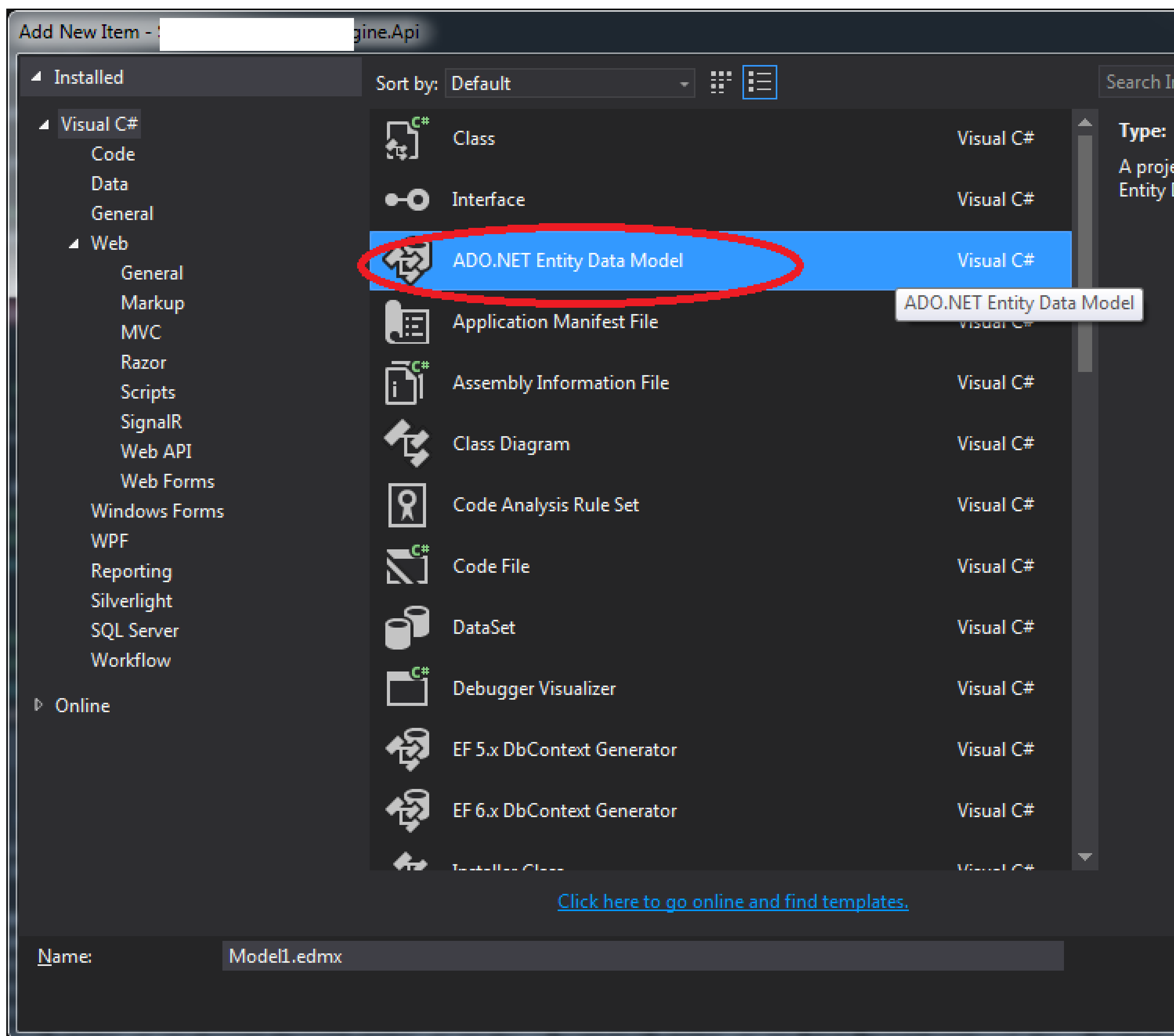
Lire Modèles de contraintes en ligne: https://riptutorial.com/fr/entity-framework/topic/4528/modeles-de-contraintes

Chapitre 16: Première génération de base de données

Examples

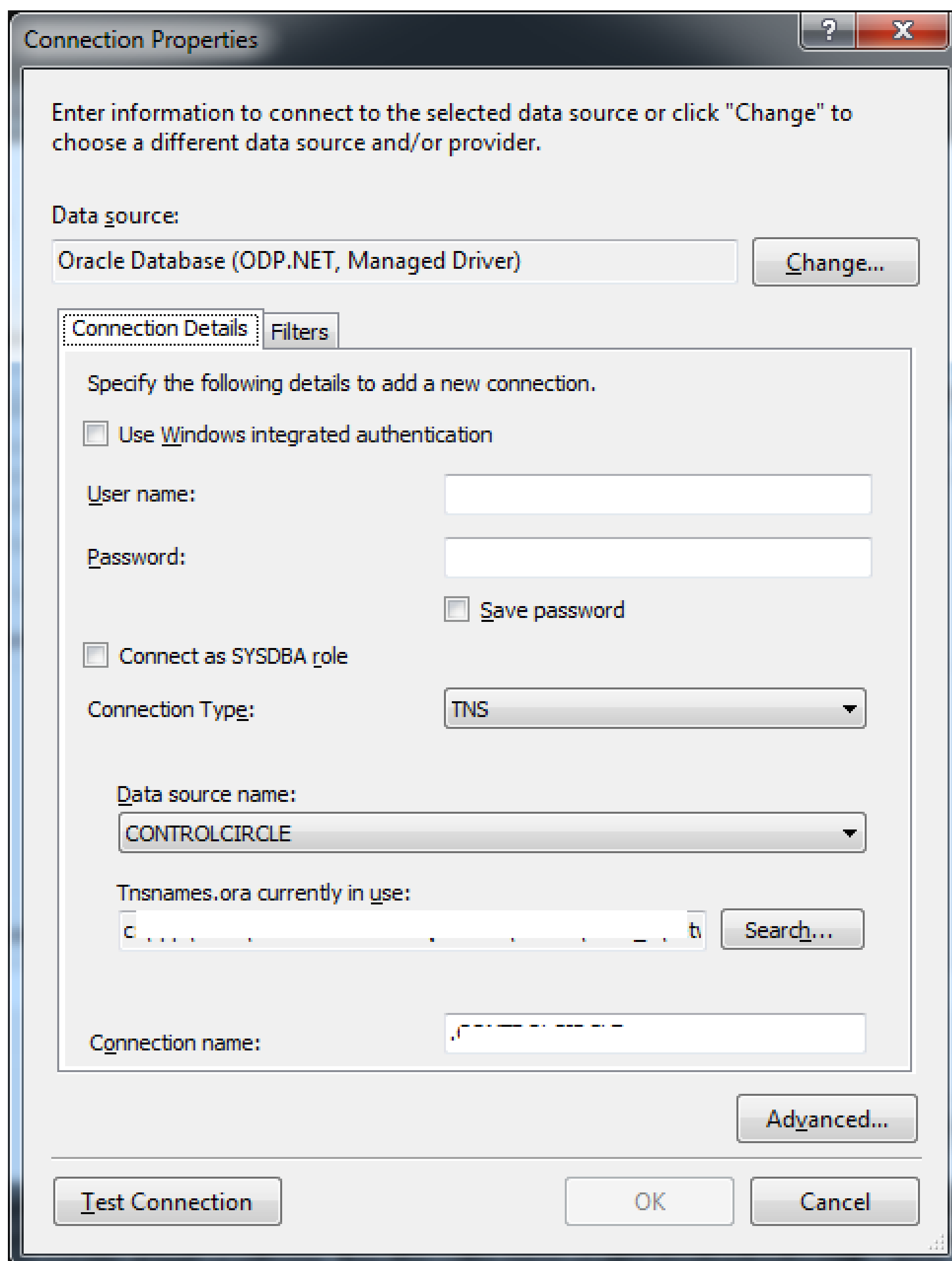
Générer un modèle à partir d'une base de données

Dans `Visual Studio` accédez à votre `Solution Explorer` puis cliquez sur `Project` vous ajouterez le modèle `Souris droite` . Choisissez le `ADO.NET Entity Data Model`



Ensuite, choisissez `Generate from database` et cliquez sur `Next` dans la fenêtre suivante, cliquez sur `New Connection...` et pointez sur la base de données à partir de laquelle vous souhaitez générer le

modèle (peut être `MSSQL` , `MySQL` ou `Oracle`).



Après cela, cliquez sur `Test Connection` pour voir si vous avez configuré la connexion correctement (n'allez pas plus loin si elle échoue ici).

Cliquez sur `Next` puis choisissez les options souhaitées (comme le style pour générer des noms d'entité ou pour ajouter des clés étrangères).

Cliquez à nouveau sur `Next` , à ce stade, vous devriez avoir le modèle généré à partir de la base de données.

Ajout d'annotations de données au modèle généré

Dans la stratégie de génération de code T4 utilisée par Entity Framework 5 et versions ultérieures, les attributs d'annotation de données ne sont pas inclus par défaut. Pour inclure des annotations de données sur certaines propriétés de chaque régénération de modèle, ouvrez le fichier de

modèle inclus avec EDMX (avec l'extension `.tt`), puis ajoutez une instruction `using` sous la méthode `UsingDirectives` comme ci-dessous:

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>
(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code);
#>
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
using System.ComponentModel.DataAnnotations;  // --> add this line
```

Par exemple, supposons que le modèle inclue `KeyAttribute` qui indique une propriété de clé primaire. Pour insérer `KeyAttribute` automatiquement lors de la régénération du modèle, recherchez une partie du code contenant `codeStringGenerator.Property` comme ci-dessous:

```
var simpleProperties = typeMapper.GetSimpleProperties(entity);
    if (simpleProperties.Any())
    {
        foreach (var edmProperty in simpleProperties)
        {
#>
    <#=codeStringGenerator.Property(edmProperty)#>
<#
        }
    }
```

Ensuite, insérez une condition if pour vérifier la propriété clé comme ceci:

```
var simpleProperties = typeMapper.GetSimpleProperties(entity);
    if (simpleProperties.Any())
    {
        foreach (var edmProperty in simpleProperties)
        {
             if (ef.IsKey(edmProperty)) {
#>    [Key]
<#      }
#>
    <#=codeStringGenerator.Property(edmProperty)#>
<#
        }
    }
```

En appliquant les modifications ci-dessus, toutes les classes de modèle générées auront `KeyAttribute` sur leur propriété de clé primaire après la mise à jour du modèle à partir de la base de données.

Avant

```
using System;

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

```
}
```

Après

```
using System;
using System.ComponentModel.DataAnnotations;

public class Example
{
    [Key]
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Lire Première génération de base de données en ligne: https://riptutorial.com/fr/entity-framework/topic/4414/premiere-generation-de-base-de-donnees

Chapitre 17: Relation de correspondance avec Entity Framework Code First: un à plusieurs et plusieurs à plusieurs

Introduction

La rubrique explique comment mapper des relations un à plusieurs et plusieurs à plusieurs à l'aide du code Entity Framework en premier.

Examples

Cartographier un à plusieurs

Alors disons que vous avez deux entités différentes, quelque chose comme ceci:

```
public class Person
{
   public int PersonId { get; set; }
   public string Name { get; set; }
}

public class Car
{
   public int CarId { get; set; }
   public string LicensePlate { get; set; }
}

public class MyDemoContext : DbContext
{
   public DbSet<Person> People { get; set; }
   public DbSet<Car> Cars { get; set; }
}
```

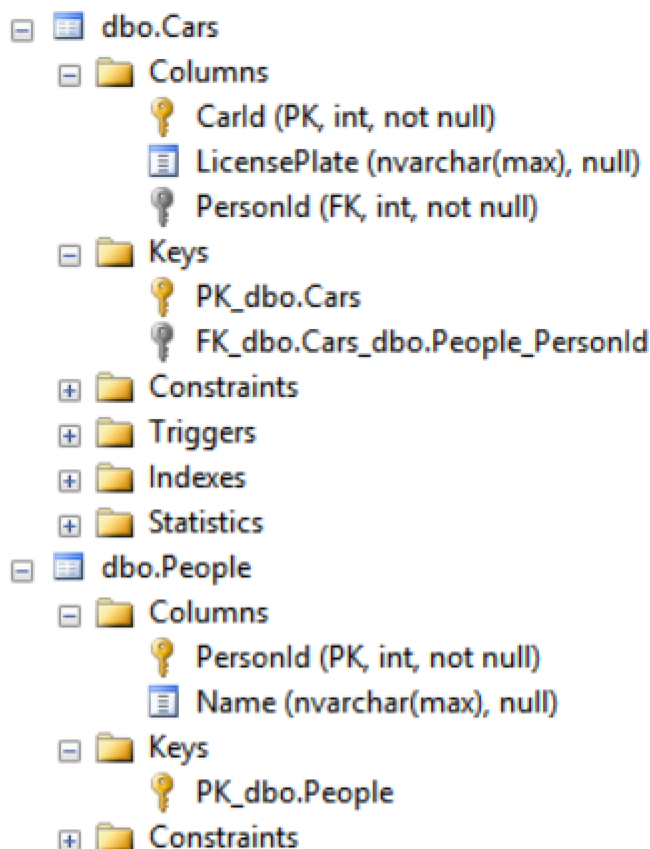
Et vous voulez établir une relation un-à-plusieurs entre eux, c'est-à-dire qu'une personne peut avoir zéro, une ou plusieurs voitures et qu'une voiture appartient à une seule personne. Chaque relation est bidirectionnelle, donc si une personne a une voiture, la voiture appartient à cette personne.

Pour ce faire, modifiez simplement vos classes de modèle:

```
public class Person
{
   public int PersonId { get; set; }
   public string Name { get; set; }
   public virtual ICollection<Car> Cars { get; set; } // don't forget to initialize (use
HashSet)
}
```

```
public class Car
{
   public int CarId { get; set; }
   public string LicensePlate { get; set; }
   public int PersonId { get; set; }
   public virtual Person Person { get; set; }
}
```

Et c'est tout :) Vous avez déjà votre relation établie. Dans la base de données, cela est bien sûr représenté par des clés étrangères.



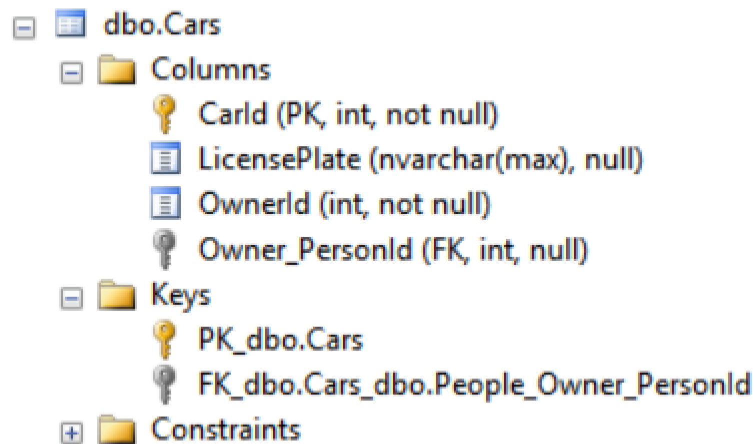
Cartographier un à plusieurs: contre la convention

Dans le dernier exemple, vous pouvez voir que EF indique quelle colonne est la clé étrangère et où doit-elle pointer. Comment? En utilisant des conventions. Avoir une propriété de type `Person` nommée `Person` avec une propriété `PersonId` conduit EF à conclure que `PersonId` est une clé étrangère et pointe vers la clé primaire de la table représentée par le type `Person` .

Mais que faire si vous *deviez* changer *PersonId* en *OwnerId* et *Person* en *Propriétaire* dans le type de **voiture** ?

```
public class Car
{
   public int CarId { get; set; }
   public string LicensePlate { get; set; }
   public int OwnerId { get; set; }
   public virtual Person Owner { get; set; }
}
```

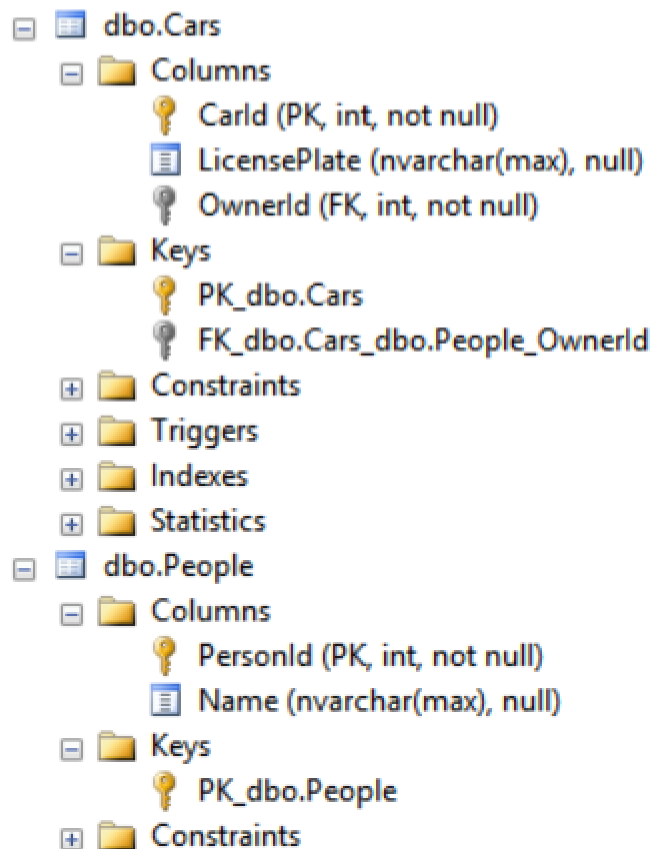
Eh bien, malheureusement, dans ce cas, les conventions ne suffisent pas à produire le schéma de base de données correct:



Pas de soucis; vous pouvez aider EF avec quelques conseils sur vos relations et vos clés dans le modèle. Configurez simplement votre type de `Car` pour utiliser la propriété `OwnerId` tant que FK. Créez une configuration de type d'entité et appliquez-la dans votre `OnModelCreating()` :

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasRequired(c => c.Owner).WithMany(p => p.Cars).HasForeignKey(c => c.OwnerId);
  }
}
```

Cela *signifie* essentiellement que `Car` a une propriété requise, `Owner` (*HasRequired ()*) et dans le type de `Owner` , la propriété `Cars` est utilisée pour renvoyer aux entités de voiture (*WithMany ()*). Et enfin la propriété représentant la clé étrangère est spécifiée (*HasForeignKey ()*). Cela nous donne le schéma que nous voulons:



Vous pouvez également configurer la relation à partir du côté `Person` :

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
  public PersonEntityTypeConfiguration()
  {
    this.HasMany(p => p.Cars).WithRequired(c => c.Owner).HasForeignKey(c => c.OwnerId);
  }
}
```

L'idée est la même, seuls les côtés sont différents (notez comment vous pouvez lire le tout: «cette personne a beaucoup de voitures, chaque voiture a un propriétaire requis»). Peu importe si vous configurez la relation du côté `Person` ou du côté `Car` . Vous pouvez même inclure les deux, mais dans ce cas, veillez à spécifier la même relation des deux côtés!

Cartographier zéro ou un à plusieurs

Dans les exemples précédents, une voiture ne peut exister sans une personne. Et si vous vouliez que la personne soit facultative du côté de la voiture? Eh bien, c'est facile, sachant comment faire un à plusieurs. Il suffit de changer le `PersonId` in `Car` pour être nullable:

```
public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public int? PersonId { get; set; }
    public virtual Person Person { get; set; }
}
```

Et puis utilisez le *HasOptional ()* (ou *WithOptional ()* , en fonction de quel côté vous faites la configuration):

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasOptional(c => c.Owner).WithMany(p => p.Cars).HasForeignKey(c => c.OwnerId);
  }
}
```

Plusieurs à plusieurs

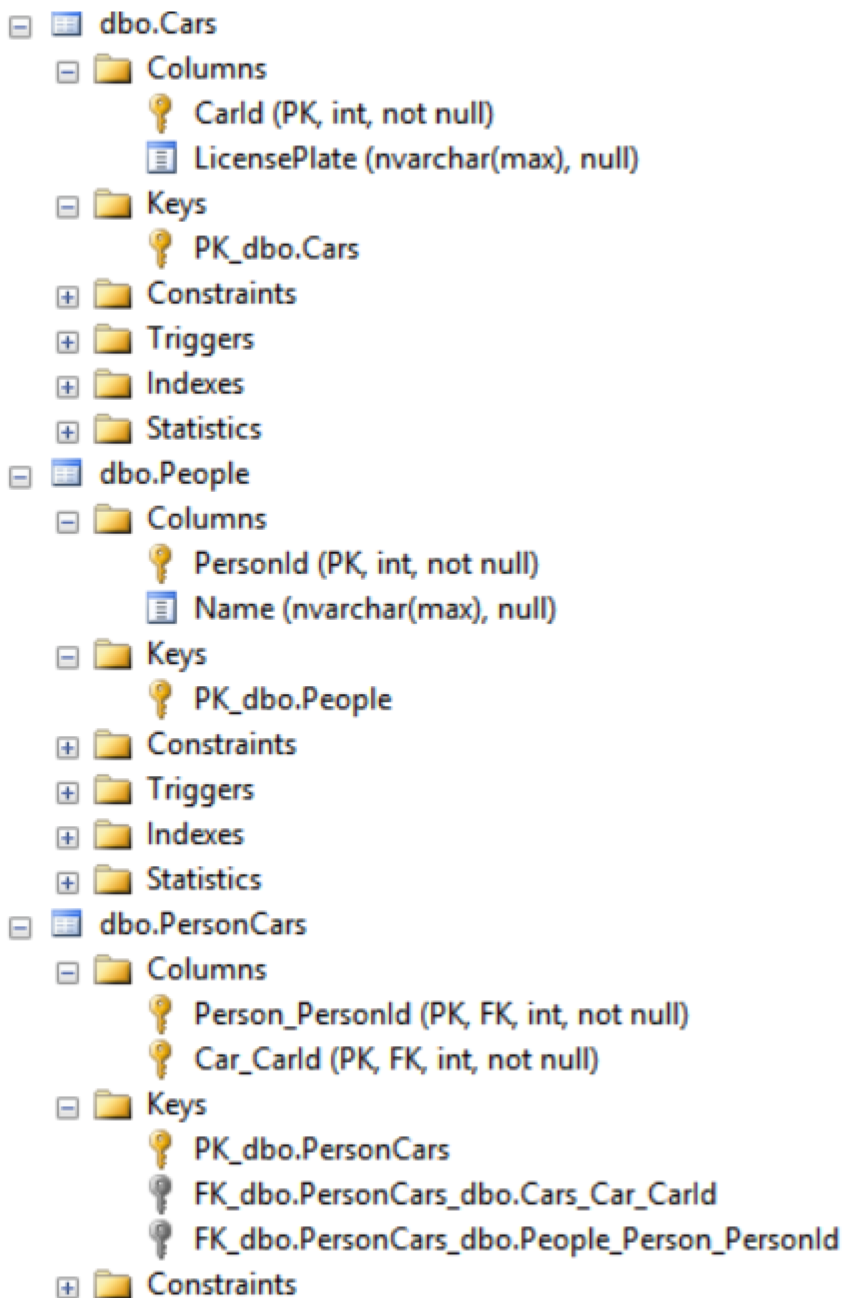
Passons à l'autre scénario, où chaque personne peut avoir plusieurs voitures et chaque voiture peut avoir plusieurs propriétaires (mais encore une fois, la relation est bidirectionnelle). C'est une relation plusieurs-à-plusieurs. Le moyen le plus simple est de laisser EF faire de la magie en utilisant des conventions.

Changez simplement le modèle comme ceci:

```
 public class Person
{
```

```
    public int PersonId { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Car> Cars { get; set; }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public virtual ICollection<Person> Owners { get; set; }
}
```

Et le schéma:



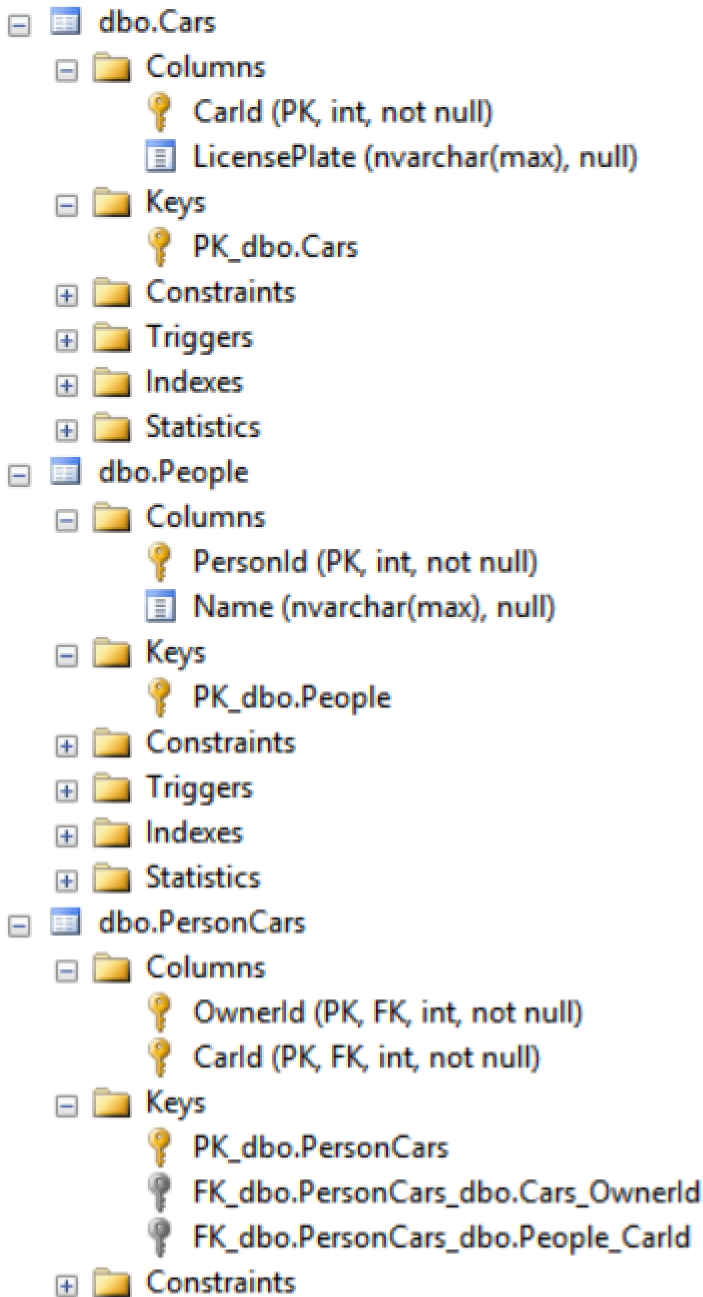
Presque parfait. Comme vous pouvez le constater, EF a reconnu la nécessité d'une table de jointure, dans laquelle vous pouvez suivre les paires de personnes.

Plusieurs à plusieurs: personnalisation de la table de jointure

Vous voudrez peut-être renommer les champs de la table de jointure pour qu'ils soient un peu plus conviviaux. Vous pouvez le faire en utilisant les méthodes de configuration habituelles (encore une fois, peu importe de quel côté vous effectuez la configuration):

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
   public CarEntityTypeConfiguration()
   {
      this.HasMany(c => c.Owners).WithMany(p => p.Cars)
          .Map(m =>
              {
                 m.MapLeftKey("OwnerId");
                 m.MapRightKey("CarId");
                 m.ToTable("PersonCars");
              }
        );
  }
}
```

Assez facile à lire même: cette voiture a beaucoup de propriétaires (*HasMany ()*), chaque propriétaire ayant plusieurs voitures (*WithMany ()*). Mappez ceci de sorte que vous mappiez la clé gauche sur OwnerId (*MapLeftKey ()*), la bonne clé pour CarId (*MapRightKey ()*) et le tout pour la table *PersonCars (ToTable ()*). Et cela vous donne exactement ce schéma:



Many-to-many: entité de jointure personnalisée

Je dois admettre que je ne suis pas vraiment fan de laisser EF inférer la table de jointure sans une entité jointe. Vous ne pouvez pas suivre les informations supplémentaires vers une association personne-voiture (disons la date à partir de laquelle elle est valide), car vous ne pouvez pas modifier la table.

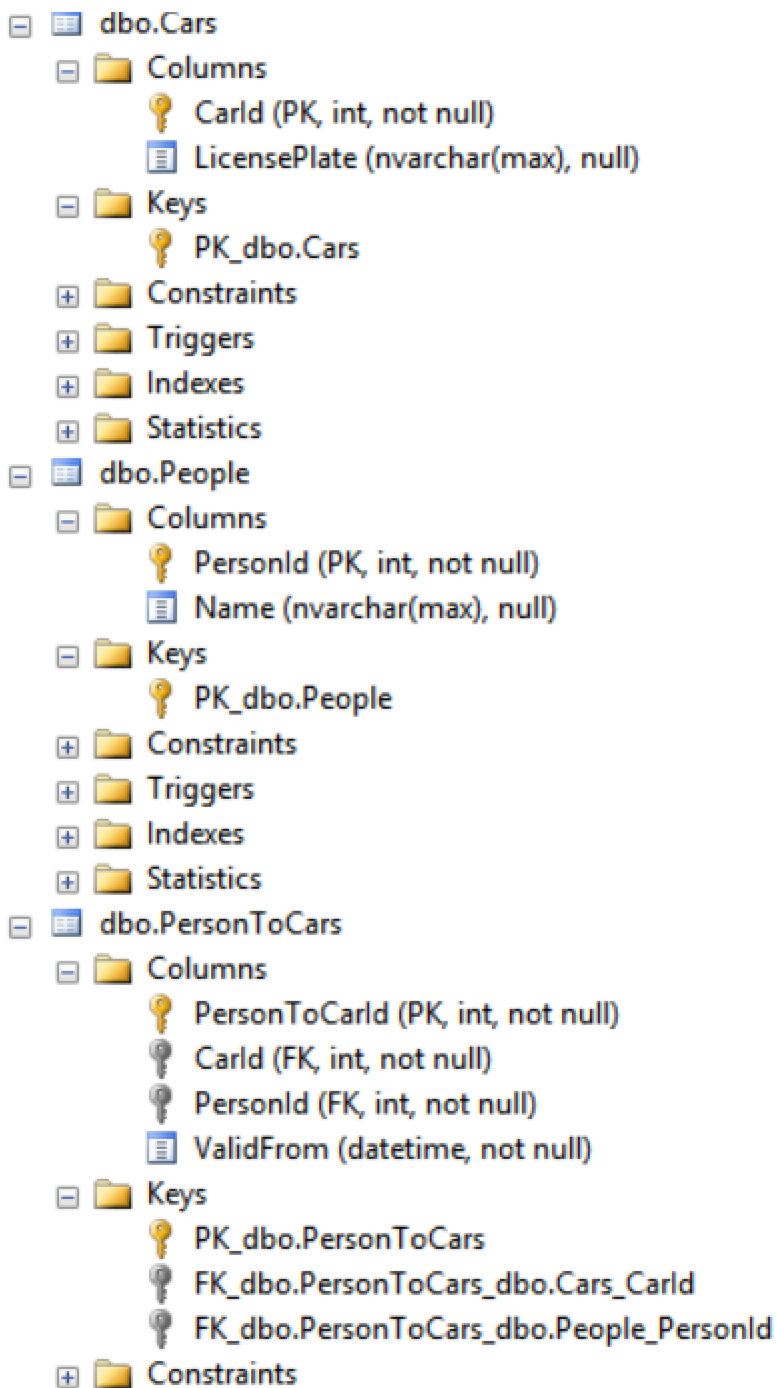
En outre, le `CarId` dans la table de jointure fait partie de la clé primaire, donc si la famille achète une nouvelle voiture, vous devez d'abord supprimer les anciennes associations et en ajouter de nouvelles. EF vous le cache, mais cela signifie que vous devez effectuer ces deux opérations au lieu d'une simple mise à jour (sans parler du fait que des insertions / suppressions fréquentes peuvent entraîner une fragmentation des index - une bonne chose est qu'il existe une solution simple).

Dans ce cas, vous pouvez créer une entité de jointure faisant référence à une voiture spécifique et

à une personne spécifique. Fondamentalement, vous considérez votre association plusieurs-à-plusieurs comme une combinaison de deux associations un-à-plusieurs:

```
public class PersonToCar
{
   public int PersonToCarId { get; set; }
   public int CarId { get; set; }
   public virtual Car Car { get; set; }
   public int PersonId { get; set; }
   public virtual Person Person { get; set; }
   public DateTime ValidFrom { get; set; }
}

public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public virtual ICollection<PersonToCar> CarOwnerShips { get; set; }
}

public class Car
{
  public int CarId { get; set; }
  public string LicensePlate { get; set; }
  public virtual ICollection<PersonToCar> Ownerships { get; set; }
}

public class MyDemoContext : DbContext
{
  public DbSet<Person> People { get; set; }
  public DbSet<Car> Cars { get; set; }
  public DbSet<PersonToCar> PersonToCars { get; set; }
}
```

Cela me donne beaucoup plus de contrôle et c'est beaucoup plus flexible. Je peux maintenant ajouter des données personnalisées à l'association et chaque association a sa propre clé primaire, de sorte que je puisse y mettre à jour la référence de la voiture ou du propriétaire.



Notez que ceci n'est qu'une combinaison de deux relations un-à-plusieurs, vous pouvez donc utiliser toutes les options de configuration présentées dans les exemples précédents.

Lire Relation de correspondance avec Entity Framework Code First: un à plusieurs et plusieurs à plusieurs en ligne: https://riptutorial.com/fr/entity-framework/topic/9413/relation-de-correspondance-avec-entity-framework-code-first--un-a-plusieurs-et-plusieurs-a-plusieurs

Chapitre 18: Relation de mappage avec Entity Framework Code First: One-to-one et variations

Introduction

Cette rubrique explique comment mapper des relations de type un à un à l'aide d'Entity Framework.

Examples

Cartographier un à zéro ou un

Alors disons encore que vous avez le modèle suivant:

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
}

public class Car
{
  public int CarId { get; set; }
  public string LicensePlate { get; set; }
}

public class MyDemoContext : DbContext
{
  public DbSet<Person> People { get; set; }
  public DbSet<Car> Cars { get; set; }
}
```

Et maintenant, vous voulez le configurer pour que vous puissiez exprimer la spécification suivante: une personne peut avoir une ou une voiture et chaque voiture appartient à une seule personne (les relations sont bidirectionnelles, donc si CarA appartient à PersonA, alors PersonA 'possède 'CarA).

Modifions donc un peu le modèle: ajoutez les propriétés de navigation et les propriétés de la clé étrangère:

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public int CarId { get; set; }
  public virtual Car Car { get; set; }
}
```

```
public class Car
{
  public int CarId { get; set; }
  public string LicensePlate { get; set; }
  public int PersonId { get; set; }
  public virtual Person Person { get; set; }
}
```

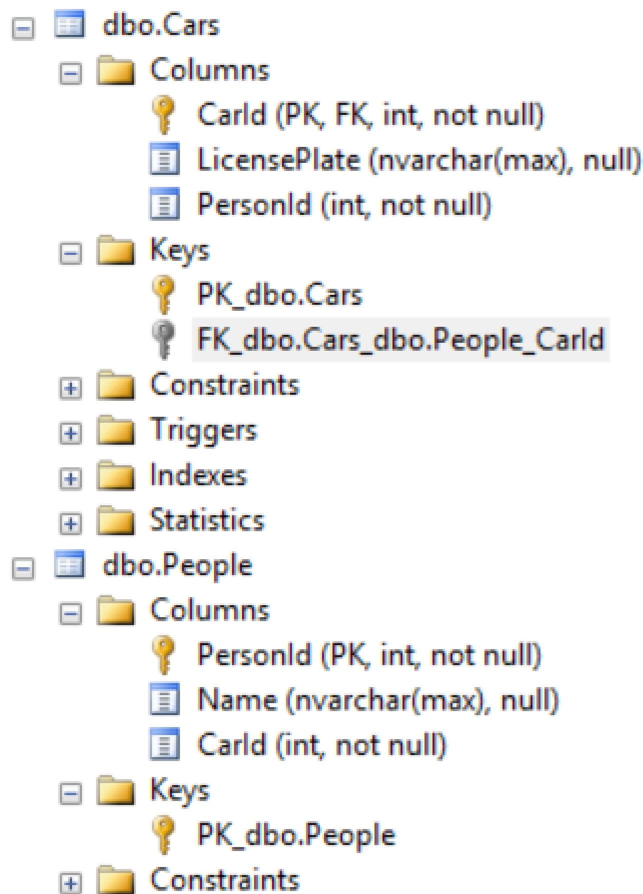
Et la configuration:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasRequired(c => c.Person).WithOptional(p => p.Car);
  }
}
```

À ce moment, cela devrait être explicite. La voiture a une personne requise (*HasRequired ()*), la personne ayant une voiture optionnelle (*WithOptional ()*). Encore une fois, peu importe de quel côté vous configurez cette relation, faites attention lorsque vous utilisez la bonne combinaison de Has / With et Required / Optional. Du côté de la `Person` , cela ressemblerait à ceci:

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
  public PersonEntityTypeConfiguration()
  {
     this.HasOptional(p => p.Car).WithOptional(c => c.Person);
  }
}
```

Maintenant, vérifions le schéma de la base de données:



Regardez attentivement: vous pouvez voir qu'il n'y a pas de FK dans `People` pour faire référence à la `Car` . En outre, le FK in `Car` n'est pas le `PersonId` , mais le `CarId` . Voici le script réel pour le FK:

```
ALTER TABLE [dbo].[Cars]  WITH CHECK ADD  CONSTRAINT [FK_dbo.Cars_dbo.People_CarId] FOREIGN
KEY([CarId])
REFERENCES [dbo].[People] ([PersonId])
```

Cela signifie donc que les propriétés de clé `CarId` et `PersonId` foregn que nous avons dans le modèle sont fondamentalement ignorées. Ils sont dans la base de données, mais ce ne sont pas des clés étrangères, comme on peut s'y attendre. En effet, les mappages individuels ne permettent pas d'ajouter le FK dans votre modèle EF. Et c'est parce que les correspondances individuelles sont assez problématiques dans une base de données relationnelle.

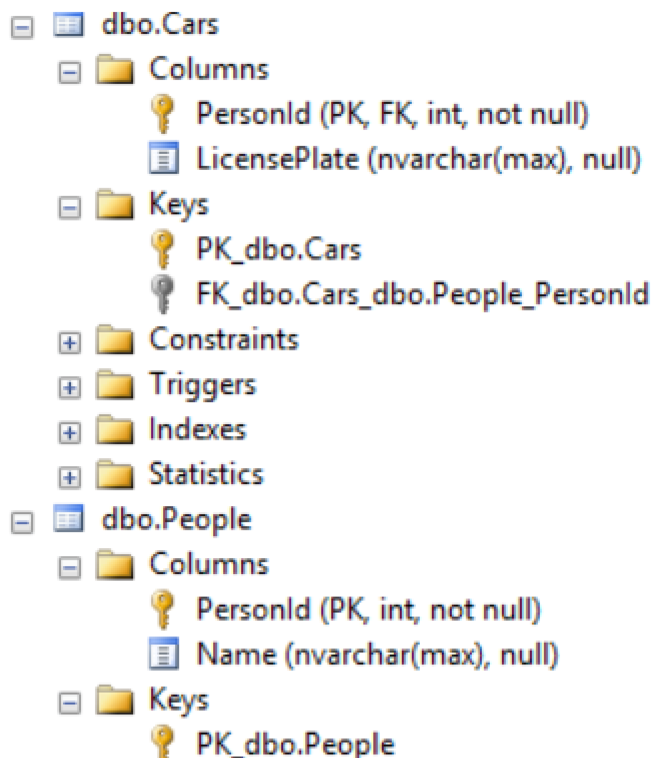
L'idée est que chaque personne peut avoir exactement une voiture et que cette voiture ne peut appartenir qu'à cette personne. Ou il peut y avoir des enregistrements de personnes auxquels aucune voiture n'est associée.

Alors, comment cela pourrait-il être représenté avec des clés étrangères? De toute évidence, il pourrait y avoir un `PersonId` in `Car` , et un `CarId` in `People` . Pour que chaque personne puisse avoir une seule voiture, `PersonId` devrait être unique en `Car` . Mais si `PersonId` est unique dans `People` , alors comment pouvez-vous ajouter deux ou plusieurs enregistrements où `PersonId` est `NULL` (plus d'une voiture sans propriétaire)? Réponse: vous ne pouvez pas (en fait, vous pouvez créer un index unique filtré dans SQL Server 2008 et versions ultérieures, mais oublions un instant cette technicité, sans parler des autres SGBDR). Sans parler du cas où vous spécifiez les deux extrémités de la relation ...

La seule manière réelle d'appliquer cette règle si les tables `People` et `Car` ont la même clé primaire (mêmes valeurs dans les enregistrements connectés). Et pour ce faire, `CarId` in `Car` doit être à la fois un PK et un FK au PK du People. Et cela complique tout le schéma. Lorsque je l'utilise, je nomme plutôt le PK / FK dans `Car PersonId` et le configure en conséquence:

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public virtual Car Car { get; set; }
}

public class Car
{
  public string LicensePlate { get; set; }
  public int PersonId { get; set; }
  public virtual Person Person { get; set; }
}

public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasRequired(c => c.Person).WithOptional(p => p.Car);
     this.HasKey(c => c.PersonId);
  }
}
```

Pas idéal, mais peut-être un peu mieux. Cependant, vous devez être vigilant lorsque vous utilisez cette solution, car elle va à l’encontre des conventions de nommage habituelles, ce qui peut vous induire en erreur. Voici le schéma généré à partir de ce modèle:



Cette relation n'est donc pas appliquée par le schéma de base de données, mais par Entity

Framework lui-même. C'est pourquoi vous devez faire très attention lorsque vous utilisez ceci, pour ne laisser personne en colère directement avec la base de données.

Cartographie individuelle

La cartographie individuelle (lorsque les deux côtés sont requis) est également une opération délicate.

Imaginons comment cela pourrait être représenté avec des clés étrangères. Encore une fois, un `CarId` dans `People` fait référence à `CarId` in `Car` et un `PersonId` in Car qui fait référence à `PersonId` dans `People` .

Maintenant, que se passe-t-il si vous souhaitez insérer un enregistrement de voiture? Pour que cela réussisse, il doit y avoir un `PersonId` spécifié dans cet enregistrement car il est requis. Et pour que ce `PersonId` soit valide, l'enregistrement correspondant dans `People` doit exister. OK, alors allons-y et insérez l'enregistrement de la personne. Mais pour que cela réussisse, un `CarId` valide doit être dans l'enregistrement personnel - mais cette voiture n'est pas encore insérée! Cela ne peut pas être parce que nous devons insérer le dossier de la personne référée en premier. Mais nous ne pouvons pas insérer l'enregistrement de la personne référée, car il renvoie à l'enregistrement de la voiture, de sorte qu'il doit être inséré en premier (réception par clé étrangère :)).

Donc, cela ne peut pas non plus être représenté comme "logique". Encore une fois, vous devez supprimer l’une des clés étrangères. Lequel vous laissez tomber est à vous. Le côté qui reste avec une clé étrangère est appelé «dépendant», le côté qui est laissé sans clé étrangère est appelé «principal». Et encore une fois, pour assurer l'unicité dans la dépendante, le PK doit être le FK, donc l'ajout d'une colonne FK et son importation dans votre modèle ne sont pas pris en charge.

Alors, voici la configuration:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
    this.HasRequired(c => c.Person).WithRequiredDependent(p => p.Car);
    this.HasKey(c => c.PersonId);
  }
}
```

À ce jour, vous devriez vraiment en avoir la logique :) Rappelez-vous juste que vous pouvez également choisir l'autre côté, faites juste attention à utiliser les versions dépendantes / principales de WithRequired (et vous devrez toujours configurer le PK dans Car).

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
  public PersonEntityTypeConfiguration()
  {
    this.HasRequired(p => p.Car).WithRequiredPrincipal(c => c.Person);
  }
}
```

Si vous vérifiez le schéma de la base de données, vous constaterez que c'est exactement la même chose que dans le cas de la solution one-to-one ou zero. C'est parce que, encore une fois, cela n'est pas appliqué par le schéma, mais par EF lui-même. Alors encore une fois, faites attention :)

Cartographier un ou un zéro ou un zéro

Et pour terminer, regardons brièvement le cas où les deux côtés sont facultatifs.

Maintenant, vous devriez vraiment vous ennuyer avec ces exemples :), donc je ne vais pas entrer dans les détails et jouer avec l'idée d'avoir deux FK-s et les problèmes potentiels et vous avertir des dangers de ne pas appliquer ces règles dans le schéma mais juste EF lui-même.

Voici la configuration à appliquer:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
    this.HasOptional(c => c.Person).WithOptionalPrincipal(p => p.Car);
    this.HasKey(c => c.PersonId);
  }
}
```

Encore une fois, vous pouvez également configurer depuis l'autre côté, faites juste attention à utiliser les bonnes méthodes :)

Lire Relation de mappage avec Entity Framework Code First: One-to-one et variations en ligne: https://riptutorial.com/fr/entity-framework/topic/9412/relation-de-mappage-avec-entity-framework-code-first--one-to-one-et-variations

Chapitre 19: Scénarios de cartographie avancés: fractionnement d'entité, fractionnement de table

Introduction

Comment configurer votre modèle EF pour prendre en charge le fractionnement d'entité ou le fractionnement de table.

Examples

Fractionnement d'entité

Alors disons que vous avez une classe d'entité comme ceci:

```
public class Person
{
   public int PersonId { get; set; }
   public string Name { get; set; }
   public string ZipCode { get; set; }
   public string City { get; set; }
   public string AddressLine { get; set; }
}
```

Et puis disons que vous voulez mapper cette entité Person en deux tables - une avec le PersonId et le Nom, et une autre avec les détails de l'adresse. Bien entendu, vous aurez également besoin du PersonId pour identifier la personne à laquelle appartient l’adresse. Donc, fondamentalement, vous voulez diviser l’entité en deux parties (voire plus). D'où le nom, entité fractionnée. Vous pouvez le faire en mappant chacune des propriétés sur une table différente:

```
public class MyDemoContext : DbContext
{
  public DbSet<Person> Products { get; set; }

  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
      modelBuilder.Entity<Person>().Map(m =>
      {
         m.Properties(t => new { t.PersonId, t.Name });
         m.ToTable("People");
      }).Map(m =>
      {
        m.Properties(t => new { t.PersonId, t.AddressLine, t.City, t.ZipCode });
        m.ToTable("PersonDetails");
      });
  }
}
```

Cela va créer deux tables: People et PersonDetails. Person a deux champs, PersonId et Name, PersonDetails a quatre colonnes, PersonId, AddressLine, City et ZipCode. Dans People, PersonId est la clé primaire. Dans PersonDetails, la clé primaire est également PersonId, mais il s'agit également d'une clé étrangère référençant PersonId dans la table Person.

Si vous interrogez le People DbSet, EF effectuera une jointure sur les PersonIds pour obtenir les données des deux tables afin de remplir les entités.

Vous pouvez également modifier le nom des colonnes:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
  modelBuilder.Entity<Person>().Map(m =>
  {
    m.Properties(t => new { t.PersonId });
    m.Property(t => t.Name).HasColumnName("PersonName");
    m.ToTable("People");
  }).Map(m =>
  {
    m.Property(t => t.PersonId).HasColumnName("ProprietorId");
    m.Properties(t => new { t.AddressLine, t.City, t.ZipCode });
    m.ToTable("PersonDetails");
  });
}
```

Cela créera la même structure de table, mais dans la table People, il y aura une colonne PersonName au lieu de la colonne Name, et dans la table PersonDetails, il y aura un ProprietorId au lieu de la colonne PersonId.

Fractionnement de table

Et maintenant, disons que vous voulez faire le contraire du fractionnement d'entité: au lieu de mapper une entité en deux tables, vous souhaitez mapper une table en deux entités. Cela s'appelle le fractionnement de table. Disons que vous avez une table avec cinq colonnes: PersonId, Name, AddressLine, City, ZipCode, où PersonId est la clé primaire. Et puis vous souhaitez créer un modèle EF comme celui-ci:

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public Address Address { get; set; }
}

public class Address
{
  public string ZipCode { get; set; }
  public string City { get; set; }
  public string AddressLine { get; set; }
  public int PersonId { get; set; }
  public Person Person { get; set; }
}
```

Une chose saute tout de suite: il n'y a pas de AddressId dans Address. En effet, les deux entités

sont mappées sur la même table, elles doivent donc avoir la même clé primaire. Si vous divisez des tables, vous devez vous en occuper. Ainsi, outre le fractionnement de table, vous devez également configurer l'entité Address et spécifier la clé primaire. Et voici comment:

```
public class MyDemoContext : DbContext
{
  public DbSet<Person> Products { get; set; }
  public DbSet<Address> Addresses { get; set; }

  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
     modelBuilder.Entity<Address>().HasKey(t => t.PersonId);
     modelBuilder.Entity<Person>().HasRequired(t => t.Address)
                                  .WithRequiredPrincipal(t => t.Person);

     modelBuilder.Entity<Person>().Map(m => m.ToTable("People"));
     modelBuilder.Entity<Address>().Map(m => m.ToTable("People"));
  }
}
```

Lire Scénarios de cartographie avancés: fractionnement d'entité, fractionnement de table en ligne: https://riptutorial.com/fr/entity-framework/topic/9362/scenarios-de-cartographie-avances--fractionnement-d-entite--fractionnement-de-table

Chapitre 20: Suivi ou non suivi

Remarques

Le comportement de suivi détermine si Entity Framework conservera ou non des informations sur une instance d'entité dans son outil de suivi des modifications. Si une entité est suivie, toutes les modifications détectées dans l'entité seront conservées dans la base de données pendant `SaveChanges()` .

Examples

Suivi des requêtes

- Par défaut, les requêtes renvoyant des types d'entités sont **suivies**
- Cela signifie que vous pouvez apporter des modifications à ces instances d'entité et que ces modifications sont conservées par `SaveChanges()`

Exemple :

- La modification de la notation du `book` sera détectée et persistante dans la base de données pendant `SaveChanges()` .

```
using (var context = new BookContext())
{
  var book = context.Books.FirstOrDefault(b => b.BookId == 1);
  book.Rating = 5;
  context.SaveChanges();
}
```

Requêtes sans suivi

- Aucune requête de suivi n'est utile lorsque les résultats sont utilisés dans un scénario en `read-only`
- Ils sont `quicker to execute` car il n'est pas nécessaire de configurer les informations de suivi des modifications.

Exemple :

```
using (var context = new BookContext())
{
    var books = context.Books.AsNoTracking().ToList();
}
```

Avec EF Core 1.0, vous pouvez également modifier le comportement de suivi par défaut au niveau de l' `context instance` .

Exemple :

```
using (var context = new BookContext())
{
    context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

    var books = context.Books.ToList();
}
```

Suivi et projections

- Même si le type de résultat de la requête n'est pas un type d'entité, si le résultat `contains entity` types d' `contains entity` ils seront toujours `tracked by default`

Exemple :

- Dans la requête suivante, qui renvoie un `anonymous type` , les instances de `Book` dans le jeu de résultats `will be tracked`

```
using (var context = new BookContext())
{
   var book = context.Books.Select(b => new { Book = b, Authors = b.Authors.Count() });
}
```

- Si le jeu de résultats `does not` contient aucun type d' `entity` , `no tracking` n'est effectué

Exemple :

- Dans la requête suivante, qui renvoie un `anonymous type` avec certaines des valeurs de l'entité (mais `no instances` du type d' `entity` réel), **aucun suivi n'est** effectué.

```
using (var context = new BookContext())
{
   var book = context.Books.Select(b => new { Id = b.BookId, PublishedDate = b.Date });
}
```

Lire Suivi ou non suivi en ligne: https://riptutorial.com/fr/entity-framework/topic/6836/suivi-ou-non-suivi

Chapitre 21: Techniques d'optimisation dans EF

Examples

Utiliser AsNoTracking

Mauvais exemple:

```
var location =  dbContext.Location
                     .Where(l => l.Location.ID == location_ID)
                     .SingleOrDefault();

return location;
```

Comme le code ci-dessus renvoie simplement une entité sans modification ni ajout, nous pouvons éviter le suivi des coûts.

Bon exemple:

```
var location =  dbContext.Location.AsNoTracking()
                     .Where(l => l.Location.ID == location_ID)
                     .SingleOrDefault();

return location;
```

Lorsque nous utilisons la fonction `AsNoTracking()` nous indiquons explicitement à Entity Framework que les entités ne sont pas suivies par le contexte. Cela peut être particulièrement utile lors de la récupération de grandes quantités de données à partir de votre magasin de données. Si vous souhaitez apporter des modifications aux entités non suivies, n'oubliez pas de les joindre avant d'appeler `SaveChanges` .

Chargement des données requises uniquement

Un problème souvent rencontré dans le code est le chargement de toutes les données. Cela augmentera considérablement la charge sur le serveur.

Disons que j'ai un modèle appelé "location" qui contient 10 champs, mais tous les champs ne sont pas requis en même temps. Disons que je ne veux que le paramètre 'LocationName' de ce modèle.

Mauvais exemple

```
var location =  dbContext.Location.AsNoTracking()
                         .Where(l => l.Location.ID == location_ID)
                         .SingleOrDefault();
```

```
return location.Name;
```

Bon exemple

```
var location =  dbContext.Location
                          .Where(l => l.Location.ID == location_ID)
                          .Select(l => l.LocationName);
                          .SingleOrDefault();

return location;
```

Le code dans le "bon exemple" ne récupérera que "LocationName" et rien d'autre.

Notez que puisque aucune entité n'est matérialisée dans cet exemple, `AsNoTracking()` n'est pas nécessaire. Il n'y a rien à suivre de toute façon.

Récupérer plus de champs avec les types anonymes

```
var location = dbContext.Location
                    .Where(l => l.Location.ID == location_ID)
                    .Select(l => new { Name = l.LocationName, Area = l.LocationArea })
                    .SingleOrDefault();

return location.Name + " has an area of " + location.Area;
```

Identique à l'exemple précédent, seuls les champs 'LocationName' et 'LocationArea' seront extraits de la base de données, le type anonyme peut contenir autant de valeurs que vous le souhaitez.

Exécutez les requêtes dans la base de données lorsque cela est possible, pas en mémoire.

Supposons que nous voulons compter le nombre de comtés au Texas:

```
var counties = dbContext.States.Single(s => s.Code == "tx").Counties.Count();
```

La requête est correcte mais inefficace. `States.Single(…)` charge un état depuis la base de données. Ensuite, `Counties` charge les 254 comtés avec tous leurs champs dans une seconde requête. `.Count()` est ensuite effectué *en mémoire* sur la collection `Counties` chargée.
Nous avons chargé beaucoup de données dont nous n'avons pas besoin et nous pouvons faire mieux:

```
var counties = dbContext.Counties.Count(c => c.State.Code == "tx");
```

Ici, nous ne faisons qu'une seule requête, qui en SQL se traduit par un compte et une jointure. Nous ne renvoyons que le compte de la base de données - nous avons enregistré les lignes, les champs et la création d’objets.

Il est facile de voir où la requête est faite en examinant le type de collection: `IQueryable<T>` vs `IEnumerable<T>`

.

Exécuter plusieurs requêtes asynchrones et parallèles

Lorsque vous utilisez des requêtes asynchrones, vous pouvez exécuter plusieurs requêtes en même temps, mais pas dans le même contexte. Si le temps d'exécution d'une requête est de 10 secondes, l'heure du mauvais exemple sera de 20 secondes, alors que le bon exemple sera de 10 secondes.

Mauvais exemple

```
IEnumerable<TResult1> result1;
IEnumerable<TResult2> result2;

using(var context = new Context())
{
    result1 = await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
    result2 = await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
}
```

Bon exemple

```
public async Task<IEnumerable<TResult>> GetResult<TResult>()
{
    using(var context = new Context())
    {
        return await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
    }
}



IEnumerable<TResult1> result1;
IEnumerable<TResult2> result2;

var result1Task = GetResult<TResult1>();
var result2Task = GetResult<TResult2>();

await Task.WhenAll(result1Task, result2Task).ConfigureAwait(false);

var result1 = result1Task.Result;
var result2 = result2Task.Result;
```

Désactiver le suivi des modifications et la génération de proxy

Si vous voulez simplement obtenir des données sans rien modifier, vous pouvez désactiver le suivi des modifications et la création de proxy. Cela améliorera vos performances et empêchera également le chargement paresseux.

Mauvais exemple:

```
using(var context = new Context())
{
    return await context.Set<MyEntity>().ToListAsync().ConfigureAwait(false);
}
```

Bon exemple:

```
using(var context = new Context())
{
    context.Configuration.AutoDetectChangesEnabled = false;
    context.Configuration.ProxyCreationEnabled = false;

    return await context.Set<MyEntity>().ToListAsync().ConfigureAwait(false);
}
```

Il est particulièrement courant de les désactiver dans le constructeur de votre contexte, surtout si vous souhaitez les définir dans votre solution:

```
public class MyContext : DbContext
{
    public MyContext()
        : base("MyContext")
    {
        Configuration.AutoDetectChangesEnabled = false;
        Configuration.ProxyCreationEnabled = false;
    }

    //snip
}
```

Travailler avec des entités de stub

Disons que nous avons des `Product` et des `Category` dans une relation plusieurs-à-plusieurs:

```
public class Product
{
    public Product()
    {
        Categories = new HashSet<Category>();
    }
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public virtual ICollection<Category> Categories { get; private set; }
}

public class Category
{
    public Category()
    {
        Products = new HashSet<Product>();
    }
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public virtual ICollection<Product> Products { get; set; }
}
```

Si nous voulons ajouter une `Category` à un `Product` , nous devons charger le produit et ajouter la catégorie à ses `Categories` , par exemple:

Mauvais exemple:

```
var product = db.Products.Find(1);
var category = db.Categories.Find(2);
product.Categories.Add(category);
db.SaveChanges();
```

(où `db` est une sous-classe `DbContext`).

Cela crée un enregistrement dans la table de jonction entre le `Product` et la `Category` . Cependant, cette table ne contient que deux valeurs `Id` . Charger deux entités complètes pour créer un enregistrement minuscule est un gaspillage de ressources.

Un moyen plus efficace consiste à utiliser *des entités stub* , c'est-à-dire des objets entité créés en mémoire, contenant uniquement le minimum de données, généralement une valeur `Id` . Voici à quoi ça ressemble:

Bon exemple:

```
// Create two stub entities
var product = new Product { ProductId = 1 };
var category = new Category { CategoryId = 2 };

// Attach the stub entities to the context
db.Entry(product).State = System.Data.Entity.EntityState.Unchanged;
db.Entry(category).State = System.Data.Entity.EntityState.Unchanged;

product.Categories.Add(category);
db.SaveChanges();
```

Le résultat final est le même, mais cela évite deux allers-retours à la base de données.

Prévenir les doublons

Si vous souhaitez vérifier si l'association existe déjà, une requête bon marché suffit. Par exemple:

```
var exists = db.Categories.Any(c => c.Id == 1 && c.Products.Any(p => p.Id == 14));
```

Encore une fois, cela ne chargera pas les entités complètes en mémoire. Il interroge efficacement la table de jonction et ne renvoie qu'un booléen.

Lire Techniques d'optimisation dans EF en ligne: https://riptutorial.com/fr/entity-framework/topic/2714/techniques-d-optimisation-dans-ef

Chapitre 22: Transactions

Examples

Database.BeginTransaction ()

Plusieurs opérations peuvent être exécutées sur une seule transaction afin que les modifications puissent être annulées si l'une des opérations échoue.

```
using (var context = new PlanetContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            //Lets assume this works
            var jupiter = new Planet { Name = "Jupiter" };
            context.Planets.Add(jupiter);
            context.SaveChanges();

            //And then this will throw an exception
            var neptune = new Planet { Name = "Neptune" };
            context.Planets.Add(neptune);
            context.SaveChanges();

            //Without this line, no changes will get applied to the database
            transaction.Commit();
        }
        catch (Exception ex)
        {
            //There is no need to call transaction.Rollback() here as the transaction object
            //will go out of scope and disposing will roll back automatically
        }
    }
}
```

Notez qu'il peut être une convention de développeur d'appeler explicitement `transaction.Rollback()` , car cela rend le code plus explicite. De plus, il *peut* exister des fournisseurs de requêtes (moins connus) pour Entity Framework qui `Dipsose` correctement `Dipsose` , ce qui nécessiterait également un appel explicite à `transaction.Rollback()` .

Lire Transactions en ligne: https://riptutorial.com/fr/entity-framework/topic/4944/transactions

Chapitre 23: Types complexes

Examples

Code premiers types complexes

Un type complexe vous permet de mapper les champs sélectionnés d'une table de base de données dans un seul type, qui est un enfant du type principal.

```
[ComplexType]
public class Address
{
    public string Street { get; set; }
    public string Street_2 { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string ZipCode { get; set; }
}
```

Ce type complexe peut ensuite être utilisé dans plusieurs types d'entités. Il peut même être utilisé plus d'une fois dans le même type d'entité.

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    ...
    public Address ShippingAddress { get; set; }
    public Address BillingAddress { get; set; }
}
```

Ce type d'entité serait alors stocké dans une table de la base de données qui ressemblerait à ceci.

Id (PK, int, not null)
Name (varchar(max), null)
ShippingAddress_Street (varchar(max), null)
ShippingAddress_Street_2 (varchar(max), null)
ShippingAddress_City (varchar(max), null)
ShippingAddress_State (varchar(max), null)
ShippingAddress_ZipCode (varchar(max), null)
BillingAddress_Street (varchar(max), null)
BillingAddress_Street_2 (varchar(max), null)
BillingAddress_City (varchar(max), null)
BillingAddress_State (varchar(max), null)
BillingAddress_ZipCode (varchar(max), null)

Bien entendu, dans ce cas, une association 1: n (Customer-Address) serait le modèle préféré, mais l'exemple montre comment des types complexes peuvent être utilisés.

Lire Types complexes en ligne: https://riptutorial.com/fr/entity-framework/topic/5527/types-complexes

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Entity Framework	Adil Mammadov, Community, DavidG, Eldho, Jacob Linney, Martin4ndersen, Matas Vaitkevicius, Nasreddine, NovaDev, Parth Patel, Stephen Reindl, tmg
2	.t4 templates dans entity-framework	Matas Vaitkevicius, Tetsuya Yamamoto
3	Chargement d'entités liées	Adil Mammadov, Florian Haider, Gert Arnold, hasan, Joshit, Matas Vaitkevicius, tmg
4	Code cadre d'entité en premier	Balázs Nagy, Jozef Lačný
5	Code Entité-cadre Premières migrations	CGritton, hasan, Joshit, Mostafa, RamenChef, Stephen Reindl
6	Code First - Fluent API	Adil Mammadov, Daniel Lemke, Jason Tyler, tmg
7	Code First DataAnnotations	bubi, CptRobby, Daniel A. White, Daniel Lemke, DavidG, Diego, Gert Arnold, Jozef Lačný, Mark Shevchenko, Matas Vaitkevicius , Parth Patel, Piotrek, tmg, Tushar patel
8	Conventions de code premier	MacakM, Parth Patel, Sivanantham Padikkasu, Stephen Reindl, tmg
9	Entity Framework avec SQLite	Jason Tyler
10	Entity-Framework avec Postgresql	skj123
11	État de l'entité de gestion	Gert Arnold
12	Héritage avec EntityFramework (Code First)	lucavgobbi
13	Initialiseurs de base de données	Jozef Lačný
14	Meilleures pratiques	Braiam, Mina Matta

	pour l'entité Framework (Simple & Professional)	
15	Modèles de contraintes	SOfanatic, Tushar patel
16	Première génération de base de données	Matas Vaitkevicius, Tetsuya Yamamoto
17	Relation de correspondance avec Entity Framework Code First: un à plusieurs et plusieurs à plusieurs	Akos Nagy
18	Relation de mappage avec Entity Framework Code First: One-to-one et variations	Akos Nagy
19	Scénarios de cartographie avancés: fractionnement d'entité, fractionnement de table	Akos Nagy
20	Suivi ou non suivi	hasan, Sampath, Stephen Reindl, tmg
21	Techniques d'optimisation dans EF	Amit Shahani, Anshul Nigam, DavidG, Gert Arnold, Jacob Linney, Kobi, lucavgobbi, Stephen Reindl, tmg, wertzui
22	Transactions	CptRobby, DavidG, Gert Arnold
23	Types complexes	CptRobby, Gert Arnold