Cyril Seguenot

2017

Résumé

Support d’accompagnement d’une formation à ADO.Net faite en présentiel

Formation à ADO.Net

Support de cours

Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l’auteur ou de ses ayants droit ou ayants cause est illicite selon le Code de la propriété intellectuelle (Art L 122-4) et constitue une contrefaçon réprimée par le Code pénal.

Seules sont autorisées (Art L122-5) les copies ou reproductions strictement réservées à l’usage privé du copiste et non destinées à une utilisation collective, ainsi que les analyses et courtes citations justifiées par le caractère critique, pédagogique ou d’information de l’œuvre à laquelle elles sont incorporées, sous réserve, toutefois, du respect des dispositions des articles L 122-10 à L 122-12 du même Code, relatives à la reproduction par reprographie.

©Cyril Seguenot 2017

Table des matières

[1 Gestion de la connexion 4](#_Toc500545676)

[1.1 Obtention d’une chaîne de connexion 4](#_Toc500545677)

[1.2 Stockage dans les paramètres de l’application 6](#_Toc500545678)

[1.3 Pool de connexions 6](#_Toc500545679)

[2 Construction et exécution de requêtes 7](#_Toc500545680)

[2.1 Récupération de données avec un DatReader 7](#_Toc500545681)

[2.2 Gestion de la valeur Null 8](#_Toc500545682)

[2.3 Récupération d’une valeur unique 9](#_Toc500545683)

[2.4 Insertion, mises à jour et suppression de données 10](#_Toc500545684)

[2.5 Paramètres de requêtes 10](#_Toc500545685)

[2.6 Transactions 11](#_Toc500545686)

[2.7 Requêtes de masse 12](#_Toc500545687)

[2.8 Utilisation d’autres SGBD 17](#_Toc500545688)

[3 Le DataSet 18](#_Toc500545689)

[3.1 Présentation 18](#_Toc500545690)

[3.2 Création d’un DataSet 19](#_Toc500545691)

[3.3 Ajout, modification et suppression de données 20](#_Toc500545692)

[3.4 Assistant Source de données 22](#_Toc500545693)

[4 Affichage et saisie des données 24](#_Toc500545694)

[5 Introduction à LINQ To SQL 29](#_Toc500545695)

[5.1 Présentation 29](#_Toc500545696)

[5.2 Mappage table - entité 30](#_Toc500545697)

[5.3 Requête avec extraction différée des données 31](#_Toc500545698)

[5.4 Extraction immédiate des données 32](#_Toc500545699)

[5.5 Requête sur des tables en relation 32](#_Toc500545700)

[5.6 Génération automatique du modèle 34](#_Toc500545701)

[6 Entity Framework 36](#_Toc500545702)

[6.1 Présentation 36](#_Toc500545703)

[6.2 Les quatre approches possibles 36](#_Toc500545704)

[6.3 Création du modèle dans l’approche Database First 37](#_Toc500545705)

[6.4 Création du modèle dans l’approche Code First 42](#_Toc500545706)

[6.5 Personnalisation du modèle avec des attributs 45](#_Toc500545707)

[6.6 Personnalisation du modèle avec l’API Fluent 46](#_Toc500545708)

[6.7 Chargement de données 48](#_Toc500545709)

[6.8 Ajout, modification et suppression de données 51](#_Toc500545710)

[6.9 Enregistrement des modifications 52](#_Toc500545711)

[6.10 Annulation des modifications 53](#_Toc500545712)

[6.11 Exécution de requêtes SQL 54](#_Toc500545713)

[6.12 Gestion des conflits 55](#_Toc500545714)

[6.13 Rafraîchissement des données 58](#_Toc500545715)

[6.14 Cas particuliers 59](#_Toc500545716)

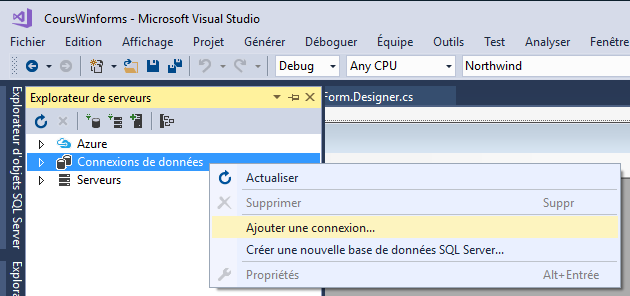
[7 Références bibliographiques 59](#_Toc500545717)

# Gestion de la connexion

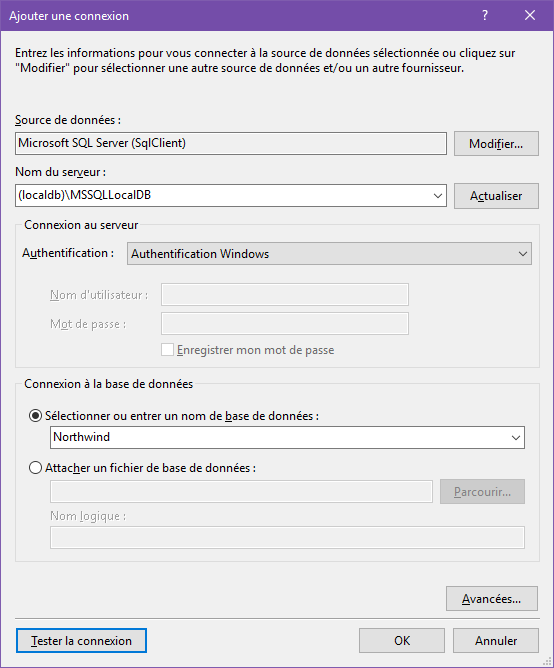
## Obtention d’une chaîne de connexion

Pour qu’une application puisse travailler avec une base de données, elle doit s’y connecter. Il faut donc créer une connexion. Pour cela :

* Ouvrir le panneau « explorateur de serveurs » de Visual Studio
* Dans le menu contextuel de la branche « Connexion de données », cliquer sur « Ajouter une connexion »



Ceci ouvre la fenêtre suivante, dans laquelle il faut sélectionner l’instance SQL Server, et la base que l’on souhaite utiliser.



Nous avons sélectionné ici l’instance SQL Server « (localDB)MSSQLLocalDB » avec authentification Windows, ainsi que la base de données Northwind disponible sur cette instance.

Le bouton Tester la connexion permet de s’assurer que la connexion est valide avant d’aller plus loin.

De retour dans le panneau « Explorateur de serveurs », nous pouvons voir la connexion créée :

|  |  |
| --- | --- |
|  | Dans ce panneau, le menu contextuel de chaque connexion permet d’ouvrir ou fermer la connexion, de la modifier, ou d’afficher ses propriétés.  Dans les propriétés, on peut notamment consulter la chaîne de connexion.  Une connexion peut bien entendu être utilisée dans plusieurs projets différents. |

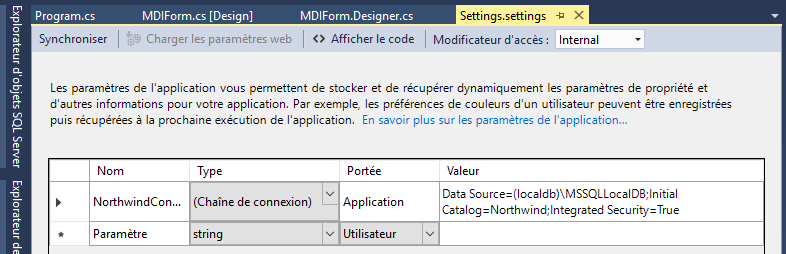
En affichant les propriétés, de la connexion, on a accès à la chaîne de connexion, qui ressemble à ceci :

Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Northwind;Integrated Security=True

C’est cette chaîne que nous utiliserons dans l’application pour nous connecter à la base et exécuter des requêtes.

## Stockage dans les paramètres de l’application

Afin de pouvoir accéder à la chaîne de connexion facilement et à de multiples endroits dans le code, il faut la stocker dans les paramètres de l’application. Pour cela, ouvrir la liste des paramètres en double-cliquant sur « Settings.settings » dans le dossier « Properties » :



Dans cette liste, ajouter une entrée de type « Chaine de connexion » et saisir la valeur obtenue précédemment depuis l’explorateur de serveur.

## Pool de connexions

Les connexions de base de données sont créées et contenues dans un pool. Lorsque l’application nécessite une connexion, ADO.Net extrait une connexion disponible du pool. Lorsque l’application ferme la connexion, celle-ci est rendue au pool et de nouveau disponible pour un prochain besoin. Pour des connexions gérées dans un pool, l’ouverture et la fermeture de connexion ne sont donc pas des opérations coûteuses. Fermer une connexion ne déconnecte pas de la base de données, mais retourne simplement la connexion au pool. L’ouverture d’une connexion implique juste l’obtention d’une connexion déjà ouverte du pool. Il est donc inutile, et même néfaste, de garder une connexion plus longtemps qu’on n’en a réellement besoin. Il faut ouvrir la connexion juste avant l’exécution de la commande, et la fermer juste après.

# Construction et exécution de requêtes

## Récupération de données avec un DatReader

L’exemple suivant montre comment mettre en œuvre une connexion, une commande et un DatReader pour récupérer des données d’une table de la base :

// Récupération de la liste des catégories de produits

public static List<Catégorie> GetCatégories()

{

   var listCategories = new List<Catégorie>();

   // On créé une commande et on définit le code sql à exécuter

   var cmd = new SqlCommand();

   cmd.CommandText = "SELECT CategoryId, CategoryName, Description FROM Categories";

   // On crée une connexion à partir de la chaîne de connexion stockée

   // dans les paramètres de l'appli

   using (var cnx = new SqlConnection(Settings.Default.NorthwindConnectionString))

   {

      // On affecte la connexion à la commande

      cmd.Connection = cnx;

      // On ouvre la connexion

      cnx.Open();

      // On exécute la commande en récupérant son résultat dans un objet SqlDataRedader

      using (SqlDataReader sdr = cmd.ExecuteReader())

      {

         // On lit les lignes de résultat une par une

         while (sdr.Read())

         {

            //...et pour chacune on crée un objet qu'on ajoute à la liste

            var cat = new Catégorie();

            cat.Id = (int)sdr["CategoryId"];

            cat.Nom = (string)sdr["CategoryName"];

            listCategories.Add(cat);

         }

      }

   }

   // Le fait d'avoir créé la connexion dans une instruction using

   // permet de fermer cette connexion automatiquement à la fin du bloc using

   return listCategories;

}

Les classes SqlConnection et SqlCommand sont définies dans l’espace de noms System.Data.SqlClient.

La classe [SqlDataReader](https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqldatareader?view=netframework-4.7) est spécialisée dans la lecture des données. Grâce à sa méthode Read, elle extrait les lignes une par une de la base de données, sans conserver de verrou sur les lignes lues. Ceci facilite beaucoup les accès concurrents de plusieurs utilisateurs à la base de données, ce qui est très important pour une application en multi-utilisateurs.

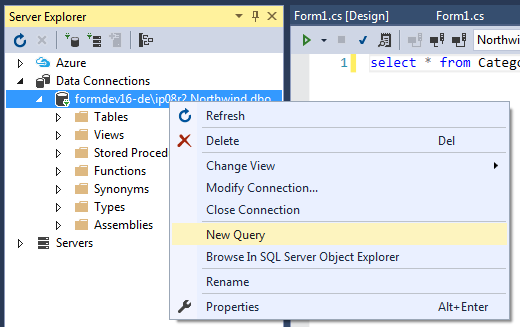
Il y a deux façons d’accéder à la valeur d’une colonne du SqlDataReader :

* Soit par son nom. La valeur obtenue est dans ce cas de type object, et il faut la transtyper
* Soit par son indice. On peut alors utiliser une des méthodes spécialisées du DataReader, telles que GetInt32, GetString, GetDateTime…etc. pour obtenir la valeur directement dans le bon type

Attention, avec la seconde méthode, si on ajoute ou si on enlève des champs dans la requête sql, il faut penser à corriger les indices, ce qui peut s’avérer fastidieux si la requête ramène beaucoup de champs.

**Astuce :**

La requête SQL peut être écrite et exécutée dans une interface spécifique de Visual Studio, semblable à celle de SQL Server Management Studio. Il suffit pour cela d’ouvrir le panneau Explorateur de Serveur, et de sélectionner « New query » dans le menu contextuel de la base :



## Gestion de la valeur Null

Dans la base, certaines colonnes de tables peuvent être nullables. Si on exécute une requête ramenant ces colonnes, les colonnes correspondantes du DataReader peuvent avoir la valeur Null. Pour gérer correctement ce cas, il faut obligatoirement comparer la valeur récupérée à DBNull.Value avant de la transtyper, sinon cela provoque une erreur. Exemple :

Supposons qu’on récupère le résultat de la requête suivante dans un SqlDataReader :

select EmployeeId, TitleOfCourtesy, LastName, FirstName, BirthDate, Region, ReportsTo  
from Employees order by 1

… la lecture du contenu du SqlDataReader devra être faite de la façon suivante :

var pers = new Personne();

pers.Id = (int)reader["EmployeeId"];

pers.Titre = (string)reader["TitleOfCourtesy"];

pers.Nom = (string)reader["LastName"];

pers.Prénom = (string)reader["FirstName"];

if (reader["BirthDate"] != DBNull.Value)

   pers.DateNais = (DateTime)reader["BirthDate"];

if (reader["Region"] != DBNull.Value)

   pers.Région = (string)reader["Region"];

if (reader["ReportsTo"] != DBNull.Value)

   pers.IdManager = (int)reader["ReportsTo"];

Dans cet exemple, pour toutes les colonnes nullables, on a testé la valeur Null avant de récupérer la valeur de la colonne.

Il est important de noter que la valeur Null du langage SQL est différente de la valeur Null du langage C#, c’est pourquoi le test précédent est fait en comparant avec la valeur DBNull.Value, qui correspond au Null du SQL.

Si on lit les valeurs du SqlDataReader en utilisant les indice de colonnes, on peut faire le test de cette façon :

if (!reader.IsDBNull(4))

   pers.DateNais = reader.GetDateTime(4);

**Types C# nullables**

En C#, tous les types références (c’est-à-dire les classes) sont nullables. Les types valeur (int, double, DateTime…etc.) peuvent être rendus nullables s’ils sont déclarés de la façon suivante :

int? Id;

Cela permet de représenter la valeur Null issue de la base, mais cela ne dispense pas de faire le test de la valeur DbNull lors de la récupération de la valeur (car les valeurs Null de C# et de SQL sont différentes).

Supposons par exemple que la propriété IdManager de la classe personne soit de type « int? ». On pourra écrire :

if (reader["ReportsTo"] != DBNull.Value)

   pers.IdManager = (int?)reader["ReportsTo"];

Si la colonne ReportsTo à la valeur Null, la propriété IdManger gardera sa valeur par défaut qui est Null.

Dans la mesure du possible, il est plus simple de définir dans la base de données les colonnes de type numérique comme non nullables, avec une valeur par défaut 0. Ceci évite d’avoir à gérer la valeur null dans le code C#, et évite également d’avoir à utiliser la fonction IsNull dans les requêtes SQL qui font des opérations mathématiques sur ces colonnes.

## Récupération d’une valeur unique

Les requêtes qui ne ramènent qu’une valeur unique (i.e. une seule colonne et une seule ligne), peuvent être exécutées de la façon suivante :

public static int GetNbProduits()

{

   var cmd = new SqlCommand();

   cmd.CommandText = @"select COUNT(\*) from Products";

   using (var cnx = new SqlConnection(Settings.Default.NorthwindConnectionString))

   {

      cmd.Connection = cnx;

      cnx.Open();

      return (int)cmd.ExecuteScalar();

   }

}

On utilise ici la méthode **ExecuteScalar** de la commande.

## Insertion, mises à jour et suppression de données

Les requêtes de type insert, update, delete ne renvoient aucun résultat, et sont exécutées de la façon suivante :

cmd.ExecuteNonQuery();

Le paragraphe qui suit donne un exemple de construction et d’exécution d’une commande de suppression.

## Paramètres de requêtes

Il peut être nécessaire de passer un ou plusieurs paramètres à une requête sql. Un cas courant est par exemple de passer en paramètre à une requête delete, l’identifiant de l’enregistrement à supprimer.

Un paramètre est représenté par un objet de type SqlParameter, et son type par une valeur de l’énumération SqlDbType. Le nom du paramètre doit être précédé de @.

Exemple :

public static void SupprimerPersonne(int idPers)

{

   var cmd = new SqlCommand();

   cmd.CommandText = @"delete from Employees where Employeeid = @id";

   // Création d'un paramètre (on précise son type, son nom et sa valeur)

   var param = new SqlParameter

   {

      SqlDbType = SqlDbType.Int,

      ParameterName = "@id",

      Value = idPers

   };

   // Ajout à la collection des paramètres de la commande

   cmd.Parameters.Add(param);

   using (var cnx = new SqlConnection(Settings.Default.NorthwindConnectionString))

   {

      cmd.Connection = cnx;

      cnx.Open();

      cmd.ExecuteNonQuery();

   }

}

La classe SqlCommand possède une propriété Parameters représentant la liste de ses paramètres. Il faut donc ajouter nos paramètres à cette liste.

Lors de l’exécution de la commande, les paramètres de la requête sql sont automatiquement remplacés par les valeurs des objet SqlParameter correspondants.

NB/ On pourrait se passer de la classe SqlParameter et construire dynamiquement la requête SQL en concaténant des chaînes (ex : string req = @"delete from Employees where Employeeid = "  + id ;). Mais ceci présente de gros inconvénients :

* Cette façon de faire est propice aux attaques de type « injection sql »
* Il faut gérer soi-même les simples côtes autour des valeurs de paramètres selon leur type, ce qui devient vite fastidieux lorsqu’il y a plusieurs paramètres
* La requête finale est très peu lisible

Il est donc fortement recommandé de ne pas utiliser cette méthode.

## Transactions

Pour s’assurer que plusieurs requêtes sont soit toutes validées, soit aucune, il faut les placer dans une transaction. L’exemple de code ci-dessous illustre la mise en œuvre d’une transaction avec ADO.Net :

public static void SupprimerCommande(int idCmde)

{

   // Préparation des commandes

   var com1 = new SqlCommand();

   com1.CommandText = @"delete from OrderDetails where OrderId = @id";

   com1.Parameters.Add(new SqlParameter {

      SqlDbType = SqlDbType.Int,   ParameterName = "@id", Value = idCmde });

   var com2 = new SqlCommand();

   com2.CommandText = @"delete from Orders where OrderId = @id";

   com2.Parameters.Add(new SqlParameter {

      SqlDbType = SqlDbType.Int,   ParameterName = "@id", Value = idCmde });

   using (var cnx = new SqlConnection(Settings.Default.NorthwindConnectionString))

   {

      // Ouverture de la connexion, et affectation aux commandes

      cnx.Open();

      com1.Connection = cnx;

      com2.Connection = cnx;

      // Transaction

      SqlTransaction tran = cnx.BeginTransaction();

      try

      {

         // Exécution des commandes, en leur affectant la transaction

         com1.Transaction = tran;

         com1.ExecuteNonQuery();

         com2.Transaction = tran;

         com2.ExecuteNonQuery();

         // Validation de la transaction

         tran.Commit();

      }

      catch(Exception)

      {

         // En cas d'erreur lors de l'exécution des commandes

         // on annule la transaction et on remonte l'erreur à l'appelant

         tran.Rollback();

         throw;

      }

   }

}

Une requête peut échouer pour différentes raisons, par exemple son code SQL n’est pas bon, ou bien certaines contraintes d’intégrité ne sont pas respectées (doublon de clé primaire, valeur de clé étrangère inexistante, colonne obligatoire non renseignée) …etc. Il est donc important de placer le code d’exécution des commandes dans un bloc try catch et d’annuler la transaction dans le bloc catch si une erreur se produit.

L’erreur ne doit pas être masquée, c’est-à-dire que l’utilisateur doit être informé que la transaction a échoué, soit par un message explicite, soit par la plante de l’application. C’est pourquoi il est important de remonter l’erreur à l’appelant avec throw après rollback de la transaction.

Une alternative possible au bloc try catch est d’utiliser une instruction using sur la transaction, comme ceci :

using (var cnx = new SqlConnection(Settings.Default.NorthwindConnectionString))

{

   ...  
   // Ouverture de la connexion, et affectation aux commandes

   cnx.Open();

   com1.Connection = cnx;

   com2.Connection = cnx;

   // Transaction

   using (SqlTransaction tran = cnx.BeginTransaction())

   {

      // Exécution des commandes, en leur affectant la transaction

      com1.Transaction = tran;

      com1.ExecuteNonQuery();

     com2.Transaction = tran;

      com2.ExecuteNonQuery();

      // Validation de la transaction

      tran.Commit();

   }

}

A la fin du using de la transaction, sa méthode Dispose est automatiquement appelée. Or celle-ci fait appel en interne à la méthode Rollback si la transaction n’a pas été validée, c’est pourquoi il n’est pas nécessaire d’appeler explicitement Rollback. L’exception éventuelle sera remontée automatiquement à l’appelant puisqu’elle n’est pas interceptée.

Cette syntaxe est plus concise, mais aussi moins explicite, car une partie du code n’est pas visible. Il convient donc de bien comprendre comment elle fonctionne si on l’utilise. Elle ne peut pas être utilisée dans le cas où on veut ajouter du code de gestion d’erreur dans la même méthode, comme par exemple loguer l’erreur dans un journal, ou bien tenter une seconde fois l’exécution de la requête.

## Requêtes de masse

Dans une application de gestion, il est fréquent de rencontrer les cas de figure suivants :

* Affichage de deux tableaux en maître-détail. Exemple : un premier tableau affiche une liste de clients et un second affiche les commandes réalisées par le client sélectionné.
* Saisie en tableau sans enregistrement entre chaque ligne : l’utilisateur saisit plusieurs lignes d’affilée, puis enregistre globalement sa saisie.

Classiquement, on peut gérer ces scénarios de façon suivante :

* A chaque clic sur une ligne du tableau maître, on exécute une requête pour ramener le contenu du tableau de détail (les commandes du client sélectionnée dans l’exemple ci-dessus)
* Lors de la saisie en tableau par l’utilisateur, on stocke les lignes dans une liste. Puis on parcourt ensuite la liste, et on exécute une requête d’insertion pour chaque ligne

Ces façons de faire fonctionnent, mais elles ne sont pas optimales d’un point de vue des performances pour la base de données. En effet, les moteurs de base de données relationnelles sont généralement conçus pour offrir les meilleures performances sur des traitements de masse. C’est-à-dire que l’exécution d’une seule requête regroupant plusieurs lignes sera plus rapide que l’exécution de plusieurs requêtes unitaires.

Nous allons voir ci-dessous comment mettre en place des traitements de masse pour récupérer, insérer, mettre à jour ou supprimer des données.

### Récupération

Il est facile de récupérer l’ensemble des données d’un maître-détail au moyen d’une requête SQL en utilisant des jointures. Le stockage de ces données dans une arborescence d’objets est par contre un peu plus délicat.

L’exemple ci-dessous montre comment récupérer une liste de régions et leurs territoires associés sous forme d’une liste d’entités Region agrégeant une liste d’entités Territoire.

Entités Region et Territoire :

public class Region

{

   public int Id { get; set; }

   public string Description { get; set; }

   public List<Territoire> Territoires { get; set; }

}

public class Territoire

{

   public string Numero { get; set; }

   public string Description { get; set; }

}

Requête pour récupérer l’ensemble des données :

public static List<Region> GetRegionsTerritoires()

{

   var listRegion = new List<Region>();

   // Obtient la liste des régions et de leurs teritoires associés

   // triés par id de région et de territoire

   string req = @"select r.RegionID, r.RegionDescription,

 t.TerritoryID, t.TerritoryDescription

                  from Region r

                  inner join Territory t on r.RegionID = t.RegionID

                  order by RegionID, TerritoryID";

   using (var cnx = new SqlConnection(Settings.Default.NorthwindConnectionString))

   {

      var cmd = new SqlCommand(req, cnx);

      cnx.Open();

      using (SqlDataReader reader = cmd.ExecuteReader())

      {

         while (reader.Read())

         {

            // Remplit la liste des régions avec leurs territoires à partir des données   
 GetTerritoiresFromDataReader(listRegion, reader);

         }

      }

   }

   return listRegion;

}

La requête ramène une ligne par territoire. Voici un extrait du résultat :

|  |  |  |  |
| --- | --- | --- | --- |
| RegionId | RegionDescription | TerritoryId | TerritoryDescription |
| 1 | Eastern | 27511 | Cary |
| 1 | Eastern | 40222 | Louisville |
| 2 | Western | 60179 | Hoffman Estates |
| 2 | Western | 60601 | Chicago |
| 2 | Western | 80202 | Denver |
| 2 | Western | 80909 | Colorado Springs |
| 3 | Northern | 03049 | Hollis |
| 3 | Northern | 03801 | Portsmouth |
| 3 | Northern | 19428 | Philadelphia |
| 3 | Northern | 44122 | Beachwood |
| 4 | Southern | 29202 | Columbia |
| 4 | Southern | 30346 | Atlanta |
| 4 | Southern | 31406 | Savannah |

Il est important que les lignes soient triées par ordre d’Id de la région puis du territoire, afin que la méthode de chargement des données dans l’arborescence d’objets fonctionne.

Cette méthode est donnée ci-dessous. Elle est appelée pour chaque ligne, et permet de remplir progressivement la liste des régions avec leurs territoires associés (liste passée en paramètre) :

private static void GetTerritoiresFromDataReader(List<Region> listRegions, SqlDataReader reader)

{

   int idRegion = (int)reader["RegionID"];

   // Si l'id de la région courante est différent de celui de la dernière région de

   // la liste, on crée un nouvel objet Region

   Region r = null;

   if (listRegions.Count == 0 || listRegions[listRegions.Count - 1].Id != idRegion)

   {

      r = new Region();

      r.Id = (int)reader["RegionID"];

      r.Description = (string)reader["RegionDescription"];

      r.Territoires = new List<Territoire>();

      listRegions.Add(r);

   }

   else r = listRegions[listRegions.Count - 1];

// Création du territoire et association à la région  
   Territoire t = new Territoire();

   t.Numero = (string)reader["TerritoryID"];

   t.Description = (string)reader["TerritoryDescription"];

   r.Territoires.Add(t);

}

La technique consiste à détecter les changements de valeur de l’Id de la région, et à chaque changement d’instancier une nouvelle région avec sa liste de territoires.

### Insertion

Pour insérer des données en masse, il est possible d’utiliser une requête du type :

insert ... select ... from MemTable

… où MemTable représente une table mémoire. Ceci ne fonctionne que si le type de la table mémoire a été préalablement défini en base au moyen d’une instruction :

create type TypeMemTable as table (...)

Le code ci-dessous montre un exemple complet d’enregistrement d’une liste de personnes dans la table Employees, en utilisant une requête d’insertion de masse.

// Insertion d'une liste de personnes en base avec requête de masse

public static void AjouterPersonnes(List<Personne> listPers)

{

   // Création d'une commande d'insertion en masse

   // Le paramètre @table contiendra les enregistrements à insérer

   var cmd = new SqlCommand();

   cmd.CommandText = @"insert Employees (LastName, FirstName, TitleOfCourtesy)

                       select Nom, Prenom, Titre from @personnes";

   // Ajout d'un paramètre de type table mémoire

   // /!\ Le type TypeTablePersonne doit être créé au préalable dans la base

   var param = new SqlParameter

   {

      SqlDbType = SqlDbType.Structured,

      ParameterName = "@personnes",

      TypeName = "TypeTablePersonne",

      Value = GetDataTableForPersonnes(listPers)

   };

   cmd.Parameters.Add(param);

   using (var cnx = new SqlConnection(Settings.Default.NorthwindConnectionString))

   {

      cmd.Connection = cnx;

      cnx.Open();

      cmd.ExecuteNonQuery();

   }

}

Le paramètre @table est de type SqlDbType.Structured. Il s’agit d’une table mémoire, dont les colonnes correspondent à celles d’un type utilisateur défini au préalable dans la base. Ce type est nommé ici TypeTablePersonne.

La méthode GetDataTableForPersonne surlignée en jaune permet de construire la table mémoire, et de la charger avec les données de la liste passée en paramètre. Voici son code :

// Création et remplissage d'une table mémoire à partir d'une liste de personnes

private static DataTable GetDataTableForPersonnes(List<Personne> listPers)

{

// Créaton d'une table mémoire

DataTable table = new DataTable();

// Création des colonnes Nom et Prenom de type chaîne et ajout à la table

var colNom = new DataColumn("Nom", typeof(string));

colNom.AllowDBNull = false;

table.Columns.Add(colNom);

var colPrénom = new DataColumn("Prenom", typeof(string));

colPrénom.AllowDBNull = false;

table.Columns.Add(colPrénom);

// Pour les colonnes nullables, on peut utiliser une syntaxe plus courte

// car AllowDBNull à la valeur True par défaut

table.Columns.Add(new DataColumn("Titre", typeof(string)));

/\* Si la colonne de clé primaire n'est pas auto-incrémentée,

on peut définir une contrainte de clé primaire sur la table

Ceci affecte automatiquement Unique = True et AllowDBNull = False

sur la ou les colonne(s) définie(s) comme clé \*/

//table.PrimaryKey = new DataColumn[] { col1, col2 };

// Chargement de la liste des personnes dans la table mémoire

foreach (var p in listPers)

{

// Création d'une ligne de table

DataRow ligne = table.NewRow();

ligne["Nom"] = p.Nom;

ligne["Prenom"] = p.Prénom;

ligne["Titre"] = p.Titre;

// Ajout de la ligne dans la table

table.Rows.Add(ligne);

}

      return table;

}

Les tables mémoires sont représentées par la classe DataTable, et leurs colonnes par la classe DataColumn, tous deux définis dans l’espace de noms System.Data.

Pour chaque colonne de la table, il faut définir au minimum un nom et un type de données (à noter que le type d’une colonne est un type .net classique, et non une valeur de l’énumération SqlDbType). On peut définir en plus des contraintes de type « unique » et « not null » à l’aide des propriétés Unique et AllowDBNull.

On peut définir la clé primaire de la table au moyen de sa propriété PrimaryKey, qui est un tableau de DataColumn. Toutes les colonnes ajoutées à ce tableau verront automatiquement leurs propriétés affectées de la façon suivante : AllowDbNull = False et Unique = True.

Enfin, voici le code SQL de création du type table dont le nom est spécifié comme valeur de la propriété TypeName du paramètre de requête :

create type TypeTablePersonne as table

(

Nom nvarchar(20) not null,

Prenom nvarchar(10) not null,

Titre nvarchar(25)

)

Ses noms de colonnes doivent correspondre aux noms des objets DataColumns créés dans la méthode GetDataTableForPersonnes.

### Mise à jour et suppression

La technique exposée précédemment peut être appliquée avec des requêtes de mise à jour ou de suppression de masse, du style :

delete e

from Employees e

inner join @table t on e.EmployeeID = t.Id

## Utilisation d’autres SGBD

Dans les paragraphes précédents, nous avons mis en œuvre les classes permettant d’exploiter une base de données SQL Server : SqlConnection, SqlCommand, SqlParameter, SqlDataReader, SqlTransaction

ADO.net fournit également des classes pour les autres fournisseurs de données principaux : Oracle, OleDB et ODBC. Les noms de ces classes sont identiques aux précédentes, en remplaçant Sql par le nom du fournisseur. Ex : OracleConnection, OleDbConnexion…

Toutes ces classes dérivent de classes ancêtres abstraites nommées DbConnection, DbCommand, DbParameter, DbDataReader, DbTransaction.

De la même façon, ADO.net fournit des énumérés pour représenter les types de chaque fournisseur :

SqlDbType, OracleType, OleDbType, OdbcType

Enfin, d’autres sociétés que Microsoft ont développé des librairies pour implémenter l’accès en .net à d’autres bases de données, telles que MySQL, SQLite…etc.

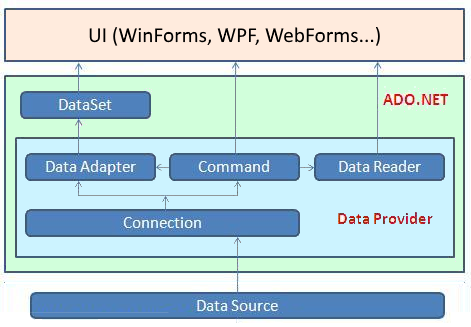
# Le DataSet

## Présentation

Dans tout ce qui précède, les méthodes de gestion des données manipulent des entités POCO ou des types simples : les méthodes de récupération renvoient des collections d’objets, et les méthodes d’insertion, mise à jour et suppression de données, travaillent avec des objets ou types simples passés en paramètres.

ADO.Net offre une autre possibilité pour manipuler les données en mémoire : le DataSet. Il permet de représenter en mémoire un jeu complet de données sous forme de tables, de relations et de contraintes. Un DataSet peut être rempli à partir d’une ou plusieurs sources de données différentes, de type bases de données relationnelles ou fichiers XML. Il permet d’avoir un modèle de programmation indépendant de la source de données manipulée.

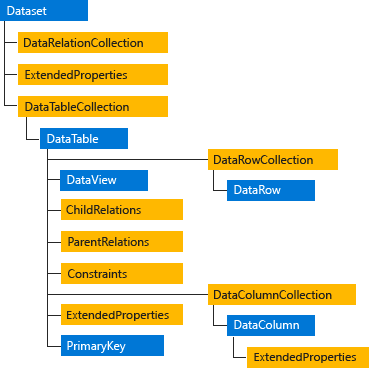
Le schéma ci-dessous illustre les différentes couches techniques et classes intervenant dans la gestion des données avec ADO.Net :



La classe DataAdpater fait le lien entre la source de données et le DataSet dans les deux sens (récupération et modification des données de la source). Un même DataSet peut être relié à plusieurs sources avec autant de DataAdapter.

Le DataSet est sérialisable, ce qui en fait un conteneur adapté au transport des données pour la communication avec des services web xml.

Le schéma ci-dessous, issu de [cette page MSDN](https://msdn.microsoft.com/fr-fr/library/zb0sdh0b(v=vs.110).aspx) montre la structure interne d’un DataSet en termes de classes :



On voit que le DataSet possède entre autres une collection de DataTable. Chaque DataTable possède une collection de DataRow pour ses lignes et une collection de DataColumn pour ses colonnes.

## Création d’un DataSet

Le code ci-dessous montre la syntaxe minimale pour remplir un DataSet constitué d’une seule table, à partir d’une commande SQL et d’un DataAdapter :

// Méthode pas utilisée par l'appli, servant juste à illustrer le cours

public static DataSet GetPersonnes()

{

   // Création d'une commande pour exécuter la requête

   var cmd = new SqlCommand(@"select EmployeeId, TitleOfCourtesy, LastName, FirstName,  
  BirthDate, ReportsTo from Employee order by 1");

   // Création d'un Dataset pour stocker le résultat

   DataSet dsPersonnes = new DataSet();

   // Création d'un DataAdpater pour faire le lien entre les deux

   SqlDataAdapter adapt = new SqlDataAdapter(cmd);

   using (var cnx = new SqlConnection(Settings.Default.Northwind2Connect))

   {

      cmd.Connection = cnx;

      // Remplissage du DataSet à partir de la source de données

      adapt.Fill(dsPersonnes);

   }

   return dsPersonnes;

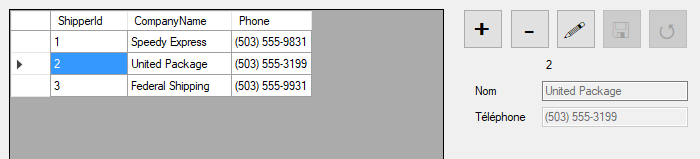
}

Le DataSet qui résulte de ce code contient une DataTable dont les noms de colonnes correspondent à ceux de la requête SQL. Notons qu’il n’est pas nécessaire d’ouvrir ni de fermer explicitement la connexion, car le DataAdapter gère lui-même cela.

## Ajout, modification et suppression de données

Le DataSet est une représentation d’un jeu de données en mémoire, qu’on peut modifier en ajoutant, modifiant ou supprimant des enregistrements. Toutes les opérations sont enregistrées par le DataSet, et peuvent ensuite être répercutées sur la source de données (base de données ou fichier XML), par l’intermédiaire d’un DataAdapter.

Pour illustrer cela, nous allons voir une partie du code permettant d’implémenter la saisie suivante :



La liste de gauche est en lecture seule et permet de naviguer dans les enregistrements. La partie droite permet d’ajouter des lignes, et de supprimer ou modifier la ligne courante.

### Couche d’accès aux données

Le code suivant montre un exemple d’implémentation d’une couche d’accès aux données, c’est à dire de la partie qui gère les interactions entre le DataSet et sa source de données (ici, une base SQL Server).

Il s’agit d’un exemple basic, faisant intervenir une seule table, et qui pour but de montrer la syntaxe de mise en oeuvre des DataSet, SqlDataAdapter et SqlCommandBuilder. Plus loin nous verrons comment utiliser l’assistant Source de données pour gérer des cas plus complexes.

public static class DAL

{

   private static SqlConnection \_connexion;

   private static SqlDataAdapter \_adapter;

   static DAL()

   {

      // Création de la connexion

      \_connexion = new SqlConnection(Settings.Default.Northwind2Connect);

      // Création d'une commande pour exécuter la requête

      var cmd = new SqlCommand(@"select ShipperId, CompanyName, Phone from Shipper");

      cmd.Connection =  \_connexion;

      // Création du DataAdpater

      \_adapter = new SqlDataAdapter();

      \_adapter.SelectCommand = cmd;

      // Génération des requêtes Insert, Update, Delete avec un constructeur de requêtes

      SqlCommandBuilder scb = new SqlCommandBuilder(\_adapter);

      scb.GetInsertCommand();

      scb.GetUpdateCommand();

      scb.GetDeleteCommand();

   }

   public static DataSet GetLivreurs()

   {

      DataSet dsLivreurs = new DataSet();

      // Cette ligne crée une DataTable dans le DataSet et affecte les propriétés

// de ses colonnes : AllowDbNull, AutoIncrement, MaxLength, RadOnly, Unique

      \_adapter.FillSchema(dsLivreurs, SchemaType.Source);

      // Remplissage du DataSet

      // La connexion est ouverte puis fermée automatiquement

      \_adapter.Fill(dsLivreurs);

      return dsLivreurs;

   }

   public static int EnregistrerModifs(DataSet ds)

   {

      return \_adapter.Update(ds);

   }

}

Analysons ce code :

Les instances de connexion et de DataAdapter sont stockées dans des variables statiques afin d’être partagées facilement entre les différentes méthodes. Elles sont initialisées dans le constructeur statique de la classe, c’est à dire avant tout appel d’une des méthodes de la classe.

La classe SqlCommandBuilder permet de générer automatiquement les requêtes d’insertion, mise à jour et suppression d’enregistrements à partir de la requête de sélection. Ceci n’est possible que si la requête Select utilise une seule table. Dans le cas contraire, il faut écrire soi-même toutes les requêtes.

La méthode FillSchema du SqlDataAdapter permet de construire la DataTable qui contiendra les données, et d’affecter les propriétés de ses colonnes, de façon à refléter les contraintes de la source de données. De cette façon, au moment de la validation de la saisie, des exceptions seront générées si les contraintes ne sont pas respectées. L’appel de FillSchema n’est pas obligatoire, mais facilite beaucoup la validation.

La méthode Fill remplit le DataSet à partir des données de la base. Elle ouvre et ferme elle-même la connexion si cette dernière n’est pas déjà ouverte.  
/!\ Si on appelle Fill sur un DataSet contenant des lignes supprimées ou modifiées, ces lignes seront perdues.

La méthode Update récupère les opérations qui ont été faites sur le DataSet, et exécute les requêtes SQL générées par le SqlCommandBuilder pour répercuter les modifications dans la base de données. Elle renvoie le nombre d’enregistrements correctement mis à jour.

### Couche cliente

Voici quelques exemples de code qui pourraient être mis en œuvre dans la couche cliente, c’est-à-dire la couche qui exploite et modifie les données du DataSet.

Là encore, il s’agit avant tout d’illustrer le fonctionnement du DataSet. Dans la pratique, on construit un formulaire de saisie de façon automatisée, grâce aux outils et composants spécifiques fournis par Visual Studio, comme nous le verrons au prochain chapitre.

// Ajout d’une ligne dans la DataTable

DataRow \_ligneCourante = \_dsLivreurs.Tables[0].NewRow();

// Modification des données de la ligne

\_ligneCourante.BeginEdit();

\_ligneCourante["CompanyName"] = "Livreurs du Nord";

\_ligneCourante["Phone"] = "0123456789";

\_ligneCourante.EndEdit();

// Lecture des données de la ligne

string nom = (string)\_ligneCourante["CompanyName"];

string telephone = (string)\_ligneCourante["Phone"];

// Suppression de la ligne

\_ligneCourante.Delete();

Quelques explications sur ce code :

Les tables d’un DataSet sont accessibles via sa collection Tables, en utilisant soit un indice, soit le nom de la table si on a fourni un nom au moment de la création du DataSet.

Lorsqu’on modifie les données d’une ligne, l’utilisation des méthodes BeginEdit et EndEdit permet de réaliser plusieurs modifications d’un coup, sans être gêné par les éventuelles contraintes qui s’appliquent sur les colonnes. Les contraintes ne sont vérifiées que lorsqu’on appelle EndEdit, et pas avant.

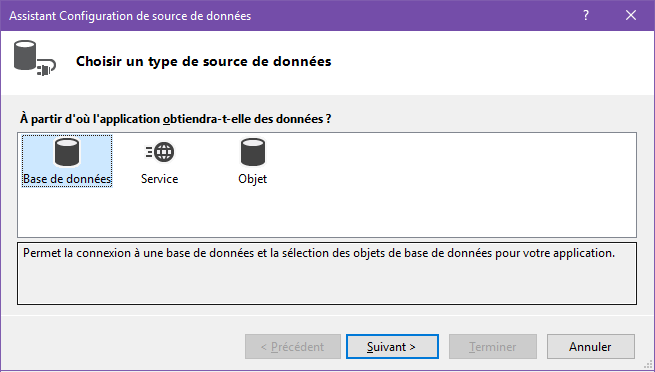
Les données d’une ligne de DataTable peuvent être lues de façon similaire à une ligne de DataReader. Les données sont retournées sous forme d’Object, qu’il faut transtyper.

## Assistant Source de données

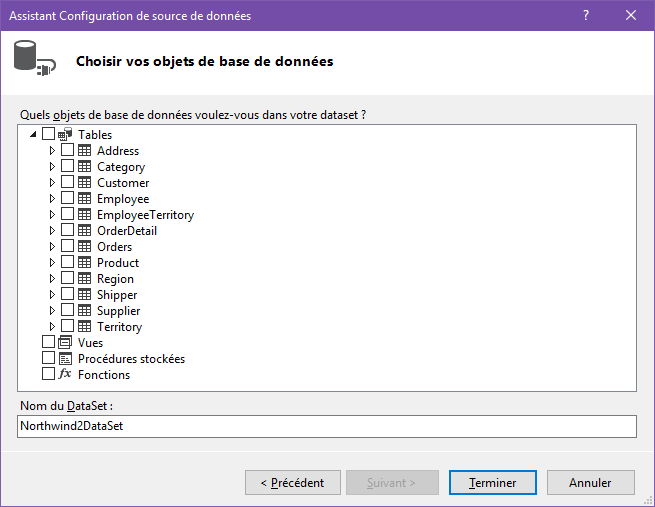
Dans l’exemple précédent, le modèle de données était basic, et nous avons pu sans peine écrire le code de la couche d’accès aux donnés. Mais lorsque le modèle est complexe et fait intervenir plusieurs tables en relation, écrire le code pour créer les DataSet et gérer l’enregistrement en base peut vite s’avérer fastidieux. C’est pourquoi Visual Studio fournit un assistant nommé « Configuration de source de données », pour générer automatiquement tout ce code.

Voici la procédure à suivre pour créer un DataSet et le code associé, à partir d’une base de données :

* Afficher le panneau « Source de données » (menu Affichage \ Autres fenêtres \ Sources de données), et cliquer sur le lien « Ajouter une nouvelle source de données »
* Ceci ouvre l’assistant « Configuration de source de données ». Sélectionner « Base de données », puis à l’étape suivante « Dataset » :



* L’étape suivante, que je ne détaillerai pas ici, permet de créer une chaîne de connexion ou d’en choisir une existante.
* L’étape suivante consiste à sélectionner les objets de base de données que nous souhaitons utiliser (tables, vues, procédures stockées, fonctions…). On peut sélectionner une ou plusieurs tables, et choisir les colonnes dans chaque table.



* Lorsqu’on clique sur le bouton Terminer, le Dataset apparaît dans le panneau sources de données, et dans l’explorateur de solutions :

|  |  |
| --- | --- |
|  |  |

Le DataSet est composé ici des tables Category, Product et Supplier. Il permettra de créer une saisie de produits, dans laquelle les catégories et fournisseurs seront proposés dans des listes déroulantes

Le DataSet peut à tout moment être modifié pour ajouter ou enlever des éléments. Il suffit pour cela de cliquer sur le bouton « Configurer le DataSet à l’aide de l’assistant » dans le panneau Sources de données.

Les fichiers générés dans le projet, et qui apparaissent dans l’explorateur de solution, contiennent des classes permettant d’utiliser le DataSet dans le code. En particulier, le fichier DataSet…Designer.cs contient des classes TableAdapter permettant de remplir les tables du DataSet et d’exécuter les requêtes CRUD dans la base. Voici un exemple d’utilisation de ces classes pour remplir la table Product du DataSet :

var ds = new DataSetProduitsFournis();

var adpat = new DataSetProduitsFournisTableAdapters.ProductTableAdapter();

adpat.Fill(ds.Product);

Dans des applications Windows Forms ou WPF, il est en plus possible de générer automatiquement une grande partie du code décrivant le visuel pour afficher/saisir les données.

# Affichage et saisie des données

Ce chapitre présente brièvement comment saisir et afficher à l’écran les données stockées dans des DataSet au moyen d’une application Windows Forms. D’autres types d’applications, telles que WPF ou UWP pourraient être utilisées.

De façon générale, on peut relier des données à un contrôle Winforms grâce au Data Binding (liaison de données). Visual Studio simplifie beaucoup la tâche en générant automatiquement des composants et leurs liaisons de données, à partir d’une source de données. Voici la démarche à suivre :

**Etape 1 : Définition de la source de données**

A l’aide de l’assistant Configuration de source de données, définir une source de données. Typiquement, ce sera un DataSet, ou bien un jeu d’entités si on utilise Entity Framework)

**Etape 2 : Paramétrage des champs**

Dans le panneau de Sources de données, pour l’objet que l’on souhaite utiliser (DataTable ou entité), définir quels seront les champs à afficher, et sous quelles formes

|  |  |
| --- | --- |
|  | Dans cet exemple, on configure la table Product pour l’affichage des données dans un formulaire.  Pour chaque champ, on peut sélectionner dans une liste déroulante le type de composant à générer pour afficher/saisir la valeur du champ.  On voit ici que :   * L’identifiant du produit sera affiché sous forme de libellé * Sa catégorie et son fournisseur seront affichés sous forme de liste déroulante * Son nom, son prix unitaire et son nombre d’unités en stock seront affichés sous forme de zones de texte * Sa disponibilité à la vente sera affichée sous forme de case à cocher * Les champs avec l’icône « sens interdit » ne seront pas affichés |

**Etape 3 : Création du formulaire**

Après avoir sélectionné le type de présentation souhaité pour la DataTable (DataGridView, ComboBox, Détails…), faire glisser la DataTable sur le formulaire. Cela crée automatiquement le visuel et les composants d’accès aux données. Dans l’exemple ci-dessous on a choisi une présentation de type Détails pour la DataTable Product :

|  |  |
| --- | --- |
|  | Composants d’accès aux donnés générés en même temps que les contrôles du formulaire : |

Les composants visuels correspondent bien aux types sélectionnés précédemment dans la table Product du DataSet. Visual Studio crée en plus automatiquement une barre de navigation comprenant des boutons de navigation, d’ajout, de suppression et d’enregistrement.

Pour comprendre la nature des 5 composants d’accès aux données générés dans le formulaire, voici une représentation des éléments constituant la chaîne de traitement des données, depuis la base jusqu’à l’affichage dans le formulaire :

Data Source

Table Adapter

Binding Source

Form

DataSet

Nous connaissons déjà les classes TableAdpater et DataSet. Elles sont générées par l’assistant de configuration de source de données.

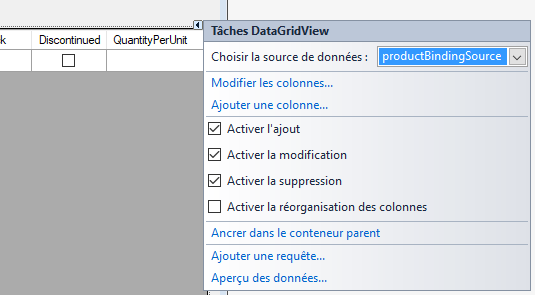
La classe BindingSource permet de faire le lien entre une DataTable du DataSet et un ou plusieurs composants d’affichage. Elle facilite la création d’une liaison de données.

Lorsqu’on a fait glisser la table Product dans un formulaire, Visual Studio a réalisé les actions suivantes :

* Il a créé les composants d’affichage/saisie dans le formulaire
* Il a instancié le DataSet (instance nommée dataSetProduitsFournis)
* Il instancié un TableAdapter (nommé productTableadapter), qui permet de remplir la DataTable à partir de la base, et inversement d’enregistrer dans la base les modifications faites sur la DataTable
* Il a instancié une Binding Source pour pouvoir relier facilement les colonnes de la DataTable Product aux composants d’affichage.
* Il a relié les composants d’affichage à la BindingSource, et la BindingSource au DataSet. Dans les propriétés de la zone de texte affichant le nom du produit, on voit par exemple que la propriété Text est liée au composant BindingSource par liaison de donnée (la propriété affiche « productBindingSource – Name »)
* Il a instancié un TableAdapterManager, qui est un composant qui gère la mise à jour des DataSet composés de plusieurs DataTable en relation.

**Etape 4 : Configuration de la DataGridView**

Lorsque les données sont affichées dans une DataGridView, il faut configurer les colonnes de celles-ci, ainsi que les actions possibles sur les données (ajout, modification, suppression). Il faut pour cela afficher le panneau de configuration en cliquant sur la petite flèche dans le coin supérieur droit du composant.



Les libellés des en-têtes de colonnes sont par défaut définis par les noms des propriétés de la classe utilisée (ici, la classe Personne). On peut spécifier d’autres libellées via le panneau de configuration ci-dessus, mais on peut aussi le faire directement dans le code de la classe correspondante, en décorant ses propriétés avec des attribut [DisplayName] :

public class Personne

{

   public int Id { get; set; }

   public string Titre { get; set; }

   public string Nom { get; set; }

   public string Prénom { get; set; }

   [DisplayName("Date de naissance")]

   public DateTime DateNais { get; set; }

   public string Région { get; set; }

   public int IdManager { get; set; }

}

Lorsque le volume de données est assez important, les performances de la DataGridView se dégradent assez rapidement, notamment à cause du fait que les lignes se redessinent sans arrêt lorsqu’on les fait défiler. Pour éviter cela, on peut placer le code suivant dans le constructeur du formulaire :

// Pour éviter que les lignes ne se redessinent lorsqu'on les fait défiler

Type dgvType = dgvTachesProd.GetType();

System.Reflection.PropertyInfo pi = dgvType.GetProperty("DoubleBuffered",

    System.Reflection.BindingFlags.Instance | System.Reflection.BindingFlags.NonPublic);

pi.SetValue(dgvTachesProd, true, null);

**Etape 5 : Configuration des listes déroulantes**

Certaines données doivent être affichées sous forme de liste déroulante pour faciliter la saisie. C’est le cas dans notre exemple, de la catégorie et du fournisseur du produit.

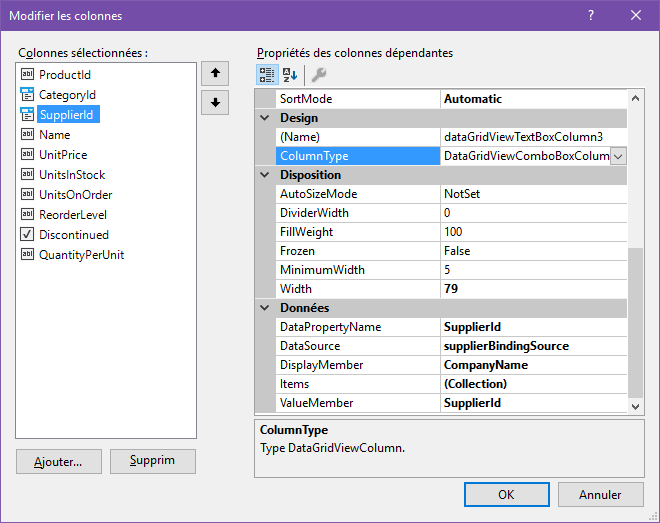
Une liste déroulante possède deux propriétés essentielles qui sont :

* DisplayMember : définit la propriété de la source à utiliser pour afficher les données dans la liste
* ValueMember : définit la propriété de la source à utiliser qui définit la valeur sélectionnée

Exemple pour la liste déroulante affichant les fournisseurs de produits : cette liste utilise comme source de donnée la DataTable Supplier. On souhaite qu’elle affiche les noms des fournisseurs, et qu’elle fournisse l’identifiant du fournisseur sélectionné lorsqu’on fait une sélection. Pour cela, on définit ses propriétés de la façon suivante :

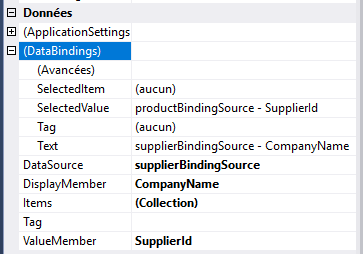
* DisplayMember = « CompagnyName »
* ValueMember = « SupplierId »

Voici un exemple de configuration de liste déroulante dans une colonne de DataGridView :



On a ici spécifié ColumnType = DataGridViewComboBoxColumn, ainsi que les propriétés de la rubrique « Données » qui définissent la source de données et les propriétés vues précédemment.

La même liste déroulante dans une présentation de type détail peut être configurée de la façon suivante :



Noter les propriétés SelectedValue et Text, qui permettent d’afficher correctement le fournisseur du produit courant.

# Introduction à LINQ To SQL

## Présentation

L’objectif de ce chapitre n’est pas de présenter LINQ To SQL dans son intégralité, mais uniquement les concepts de base, afin de se faire une idée de son fonctionnement. Pour aller plus loin, on pourra consulter les références suivantes :

* [Point d'entrée](https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/) dans docs.microsoft.com
* [Point d'entrée](https://weblogs.asp.net/dixin/Tags/LINQ%20to%20SQL) d’une série d’articles de Dixin Yan, MVP Microsoft

**Principe**

LINQ to SQL est une forme étendue de LINQ qui permet d’interagir avec une base de données relationnelle comme on le fait avec une collection d'objets en mémoire. Pour l’utiliser, il faut ajouter une référence vers l’assembly Sytem.Data.Linq.

LINQ to SQL permet de mapper le modèle de données de la base à un modèle objet constitué de classes appelées "classes de données" ou "entités". On parle de « mapping objet-relationnel ».  
La création du modèle objet peut être assez fastidieuse, c’est pourquoi Visual Studio fournit l’utilitaire « Object Relational Designer » qui facilite cette tâche.

Les requêtes LINQ sont transformées automatiquement en requêtes SQL, et leurs résultats en objets prêts à être utilisés par l’application, sans qu'on ait besoin de mettre soi-même en œuvre des SqlCommand et DataReader (ils sont créés en interne de façon transparente).

LINQ to SQL est implémenté dans l’assembly System.Data.Linq.dll du .Net Framework 3.5 ou supérieur.

**Opérations prise en charges**

LINQ To SQL permet de gérer tous les types d’opérations : Insertion, mise à jour et suppression de données

Cf. [cette page](https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/walkthrough-manipulating-data-csharp) pour voir des exemples de code.

**Limitations de LINQ To SQL**

* Utilisable uniquement avec SQL Server
* Adapté au développement rapide d’applications exécutant des requêtes simples, mais pas à des applications exécutant des requêtes complexes et/ou avec des contraintes de performances élevées.
* Le modèle objet reste très proche de la structure de la base de données
* Prend en charge de façon très limitée les modifications de structure de la base. Si la structure change, il faut régénérer une partie du modèle, et on perd les éventuelles personnalisations qu’on a faites.

## Mappage table - entité

Le mappage d’une classe (entité) sur une table de la base de données est réalisé au moyen d’attributs. Voici un exemple illustrant la syntaxe :

using System;

using System.Data.Linq.Mapping;

using System.Linq;

namespace TestLinqToSql

{

   [Table(Name = "Product")]

   public class Produit

   {

      [Column(Name="ProductId", IsPrimaryKey=true, CanBeNull=false)]

      public int Id { get; set; }

      [Column(Name="Name", CanBeNull = false)]

      public string Nom { get; set; }

      [Column(Name="SupplierId")]

      public int? IdFournisseur { get; set; }

      [Column(Name="UnitPrice")]

      public decimal? PU { get; set; }

   }

}

L’attribut [Table] décrit une table de la base de données. Il est modélisé par la classe TableAttribute.

L’attribut [Column] décrit une colonne de table. Il est modélisé par la classe [ColumnAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.data.linq.mapping.columnattribute?view=netframework-4.7), dont voici les propriétés principales :

* Name : nom de la colonne (héritée de [DataAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.data.linq.mapping.dataattribute?view=netframework-4.7)). Si ce nom est identique à celui de la propriété de l’entité, alors il peut être omis.
* Storage : nom du champ privé de la propriété, si celle-ci est déclarée de façon explicite (héritée de DataAttribute)
* CanBeNull : caractère nullable
* IsPrimaryKey : est-ce que le champ appartient à la clé primaire

Il peut créer des entités à partir des données récupérées par une requête.

Il peut également créer les tables à partir des classes de données avec DataContext.CreateDatabase.

## Requête avec extraction différée des données

Voici comment créer et exécuter une requête LINQ To SQL :

using System;

using System.Collections.Generic;

using System.Data.Linq;

using System.Linq;

namespace TestLinqToSql

{

   class Program

   {

      // Chaîne de connexion à la base

      private static string connect = @"Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Northwind2;Integrated Security=True";

      static void Main(string[] args)

      {

         // Objet à partir duquel on accède à la base en LINQ

         DataContext bdd = new DataContext(connect);

         // Table typée sur laquelle est exécutée la requête LINQ

         Table<Produit> produits = bdd.GetTable<Produit>();

         // Requête pour récupérer les produits du fournisseur dont l'Id = 1

         IQueryable<Produit> reqProduits = produits.Where(p => p.IdFournisseur == 1);

         // Exécution de la requête   
 // (une SqlCommand et un DataReader sont créés en interne)

         foreach (Produit p in reqProduits)

         {

            Console.WriteLine("{0} : {1}, {2:C2}", p.Id, p.Nom, p.PU);

         }

         Console.ReadKey();

      }

   }

}

Comme pour une requête ADO classique, on a besoin d’une chaîne de connexion à la base de données. Celle-ci est ensuite passée en paramètre à un objet DataContext qui représente la base pour LINQ.

On crée ensuite une table typée, qui représente l’ensemble d’entités sur lequel porte la requête LINQ. En effet, une requête LINQ porte toujours sur une collection d’objets en mémoire. Après construction de la requête LINQ, les données ne sont pas encore récupérées de la base. C’est pourquoi on parle d’extraction différée.

NB/ Table<T> dérive de IQueryable<T>, qui dérive elle-même de IEnumerable<T>. C’est donc un objet énumérable, c’est-à-dire parcourable par foreach.

Quand on commence le parcours avec foreach, la requête LINQ est automatiquement traduite en une requête SQL, qui est exécutée au moyen d’une SqlCommand. Les données sont récupérées dans un DataReader, et transférées dans les entités avec les méthodes GetXXX du DataReader. Tout ce mécanisme est interne à LINQ To SQL et transparent pour nous.

**Récupération d’une ligne unique**

Si on souhaite récupérer une seule ligne, il est inutile de construire une requête LINQ. Celle-ci peut être remplacée par l’appel à la méthode Single() de Table<T>. On lui passe en paramètre un prédicat sous forme d’une expression lambda, comme le montre cet exemple :

Produit vin = produits.Single(p => p.Id == 38);

Console.WriteLine("{0}, {1:C2}", vin.Nom, vin.PU);

## Extraction immédiate des données

LINQ To SQL génère une requête SQL et récupère son résultat au moyen d’un DataReader qu’il parcourt ensuite en interne lorsqu’on fait la boucle foreach. Le DataReader extrait les données de la base ligne à ligne au fur et à mesure de son parcours, ce qui permet d'avoir les données les plus récentes possibles.

Ce fonctionnement peut être gênant si la base de données est distante, c’est-à-dire accédée via une connexion réseau, car dans ce cas, une requête réseau est exécutée pour chaque ligne du DataReader. Pour éviter cela, on peut forcer l'évaluation de la requête LINQ to SQL au moment de sa création. Toutes les données seront alors mises en cache sur le serveur et récupérées en une seule requête réseau. Il suffit de forcer l’énumération du DataReader interne en appelant par exemple la méthode ToList() à la fin de la requête :

IEnumerable<Client> reqClients =   
 clients.Where(c => c.Société.Substring(0, 1) == "P").ToList();

Le résultat est alors récupéré dans une variable de type List<T> ou IEnumerable<T>, mais pas IQueryable<T>, car List<T> n’implémente pas IQueryable<T>

NB/ De la même façon, l’appel d’une fonction d’agrégation telle que Count() dans la requête, provoque son exécution immédiate.

## Requête sur des tables en relation

Dans la base Northwind, les tables Orders et Customer sont en relation 0,n – 1,1, c’est-à-dire que chaque commande possède un champ de clé étrangère représentant l’id du client ayant passé la commande.

Supposons qu’on souhaite afficher les résultats suivants :

* Le nombre de commandes de chaque client
* La liste des commandes du mois de mai 2017, avec le nom des clients correspondants

Pour cela, la relation entre les tables Orders et Customer doit être retranscrite dans le modèle objet sur les entités Commande et Client. Voici comment faire :

using System;

using System.Data.Linq;

using System.Data.Linq.Mapping;

using System.Linq;

namespace TestLinqToSql

{

   [Table(Name = "Customer")]

   public class Client

   {

      // Champs privés

      private EntitySet<Commande> \_commandes;

      // Propriétés

      [Column(IsPrimaryKey=true, Name="CustomerId", CanBeNull=false)]

      public string Id { get; set; }

[Column(Name ="CompanyName", CanBeNull=false)]

public string Société { get; set; }

      [Association(Storage = "\_commandes", OtherKey = "IdClient")]

      public EntitySet<Commande> Commandes

      {

         get { return \_commandes; }

         set { \_commandes.Assign(value); }

      }

      // Constructeur

      public Client()

      {

         \_commandes = new EntitySet<Commande>();

      }

   }

Les commandes du client sont représentées par la propriété Commandes, de type EntitySet<Commande>.

Noter que l’accesseur set utilise la méthode Assign pour affecter la valeur de cette propriété.

L’attribut [Association] spécifie comment obtenir les commandes du client :

* Sa propriété Storage identifie le champ privé utilisé par la propriété
* Sa propriété OtherKey spécifie quelle propriété de la classe Commande fait le lien avec le client (c’est-à-dire représente la clé étrangère entre Orders et Customer)

Voyons maintenant le code de l’entité Commande :

   [Table(Name = "Orders")]

   public class Commande

   {

      // Champs privés

      private EntityRef<Client> \_client;

      // Propriétés

      [Column(Name = "OrderId", IsPrimaryKey = true, CanBeNull = false)]

      public int Id { get; set; }

      [Column(Name = "CustomerId", CanBeNull = false)]

      public string IdClient { get; set; }

[Column(Name = "OrderDate", CanBeNull = false)]

public DateTime Date { get; set; }

      [Association(Storage = "\_client", ThisKey = "IdClient")]

      public Client Client

      {

         get { return \_client.Entity; }

         set { \_client.Entity = value; }

      }

      // Constructeur

      public Commande()

      {

         \_client = new EntityRef<Client>();

      }

   }

}

Le client ayant passé la commande est modélisé par la propriété Client, qui utilise en interne un champ de type EntityRef<Client>, c’est-à-dire une « référence d’entité Client ».

L’attribut [Association] spécifie comment obtenir le client ayant passé la commande :

* Sa propriété Storage identifie le champ privé utilisé par la propriété.
* Sa propriété ThisKey indique quelle propriété de la classe Commande fait le lien avec le client (c’est-à-dire représente la clé étrangère entre Orders et Customer)

La description des relations dans les entités rend désormais le requêtage LINQ simple :

// Affichage du nombre de commandes par client

private static void AfficherNbCommandesParClient()

{

   DataContext bdd = new DataContext(connect);

   Table<Client> clients = bdd.GetTable<Client>();

   IEnumerable<Client> reqClients = clients.Where(c => c.Commandes.Any());

   foreach (Client c in reqClients)

   {

      Console.WriteLine("Client {0} : {1} commandes", c.Id, c.Commandes.Count);

   }

}

// Affichage des commandes du mois de mai 2017 avec les noms des clients

private static void AfficherCommandes()

{

   DataContext bdd = new DataContext(connect);

   Table<Commande> commandes = bdd.GetTable<Commande>();

   IEnumerable<Commande> reqCommandes =

      commandes.Where(com => com.Date.Year == 2017 && com.Date.Month == 5);

   foreach (Commande c in reqCommandes)

   {

     Console.WriteLine("Commande {0} passée par le client {1}", c.Id, c.Client.Société);

   }

}

La description des relations dans les entités nous dispense de faire des jointures dans les requêtes LINQ.

## Génération automatique du modèle

Microsoft fournit deux outils permettant de générer automatiquement le modèle objet à partir d’une base de données : l’utilitaire en ligne de commande SQLMetal et le concepteur visuel objet-relationnel.

[SQLMetal](https://docs.microsoft.com/en-us/dotnet/framework/tools/sqlmetal-exe-code-generation-tool) est présent dans le répertoire :  
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.2 Tools\SqlMetal.exe"  
(chemin à adapter selon la version du SDK Windows et du framework .net utilisé).

Mais il peut être lancé plus simplement à partir de l’invite de commande « Developer Command Prompt » de Visual Studio

# Entity Framework

## Présentation

Entity Framework (EF) répond au même besoin initial que LINQ To SQL : créer un mapping objet-relationnel (O/RM) permettant d’exploiter facilement une base de données relationnelle, sans se soucier de sa structure, ni écrire de code fastidieux pour construire et exécuter des commandes SQL.

Mais EF va bien plus loin que LINQ To SQL :

* Il est compatible avec la majorité des bases de données relationnelles, et pas uniquement avec SQL Server.
* Il permet de générer un modèle conceptuel de la base de données, appelé « entity data model » personnalisable, qui protège l’application des modifications de structure de la base de données.
* Il offre 4 approches possibles, selon qu’on souhaite partir d’un jeu d’entités, d’un modèle ou d’une base de données. Il peut générer automatiquement la base de données à partir du code

Depuis sa version 6, Entity Framework n’est plus intégré au .net Framework. Il est fourni sous forme de package NuGet comprenant deux dll qui doivent être déployées avec l’application.

**Entity Framework Core** est la version à utiliser avec .Net Core, également fournie sous forme de package NuGet.

EF utilise une variante de LINQ appelée **LINQ to Entities** pour faire des requêtes sur les données.

Il y a deux couches principales dans Entity Framework :

* La couche de modélisation, composée de :
  + Un modèle conceptuel Entity Data Model, composé de types d’entités et de relations
  + Un schéma de base de données
  + Un mappage entre le modèle conceptuel et le schéma de base
* La couche objet

Le mappage permet de transformer les opérations réalisées sur les entités en requêtes SQL sur la base.

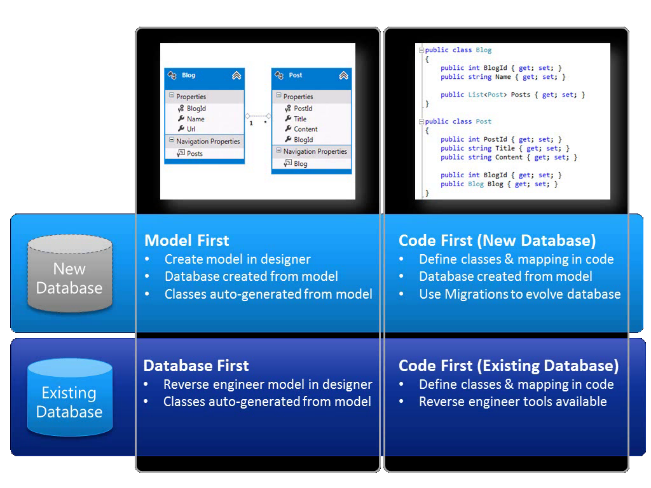
La couche objet reflète le modèle conceptuel sous forme d’objets POCO typés, directement exploitables par le développeur. Sur cette couche, on peut lier les objets à des contrôles visuels, faire des requêtes, faire un suivi des modifications des données, et transférer les modifications à la base de données.

[Point d’entrée](https://msdn.microsoft.com/en-us/library/ee712907(v=vs.113).aspx) de la documentation MSDN

[Site Entity Framework Tutorial](http://www.entityframeworktutorial.net/)

## Les quatre approches possibles

Il existe plusieurs approches possibles pour démarrer un projet, selon qu’on dispose déjà d’une base de données ou non, et selon qu’on préfère commencer par créer les entités ou par la créer la base. Le schéma ci-dessous en donne une vue synthétique :



Dans l’approche Model First, la base de données n’existe pas au départ. Avec le designer, on commence par créer un modèle, qu’on enregistre dans un fichier .edmx (Entity Data Model). Puis on génère à partir de ce modèle un script de création de base de données, ainsi que les classes POCO de la couche objet.

Dans l’approche Database First, on génère le modèle à partir d’une base de données existante, puis on génère les classes de la couche objet à partir du modèle.

Dans l’approche Code First, le modèle est défini par le code, à partir des types d’objets et des configurations supplémentaires qu’on définit. Si la base de données existe déjà, le modèle est mappé dessus. Sinon, la base est générée à partir du modèle.

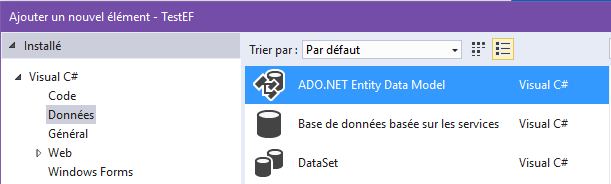
Nous détaillerons dans ce cours les approches avec base de données existante.

## Création du modèle dans l’approche Database First

Nous allons voir ici comment générer le modèle conceptuel de données et son mappage avec les tables d’une base existante. Ceci peut être réalisé dans n’importe quel type d’application (Console, Winform, WPF, ASP.Net…). Les représentations des modèles sont par contre différentes dans Entity Framework 6 et Entity Framework Core, c’st pourquoi nous allons détailler les deux.

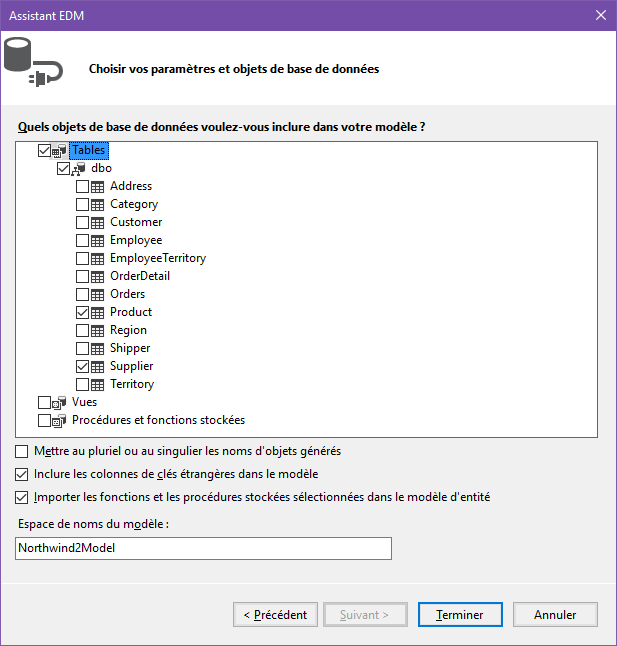
### Avec Entity Framework 6

Dans le projet Visual Studio, ajouter un fichier de type ADO.NET Entity Data Model :



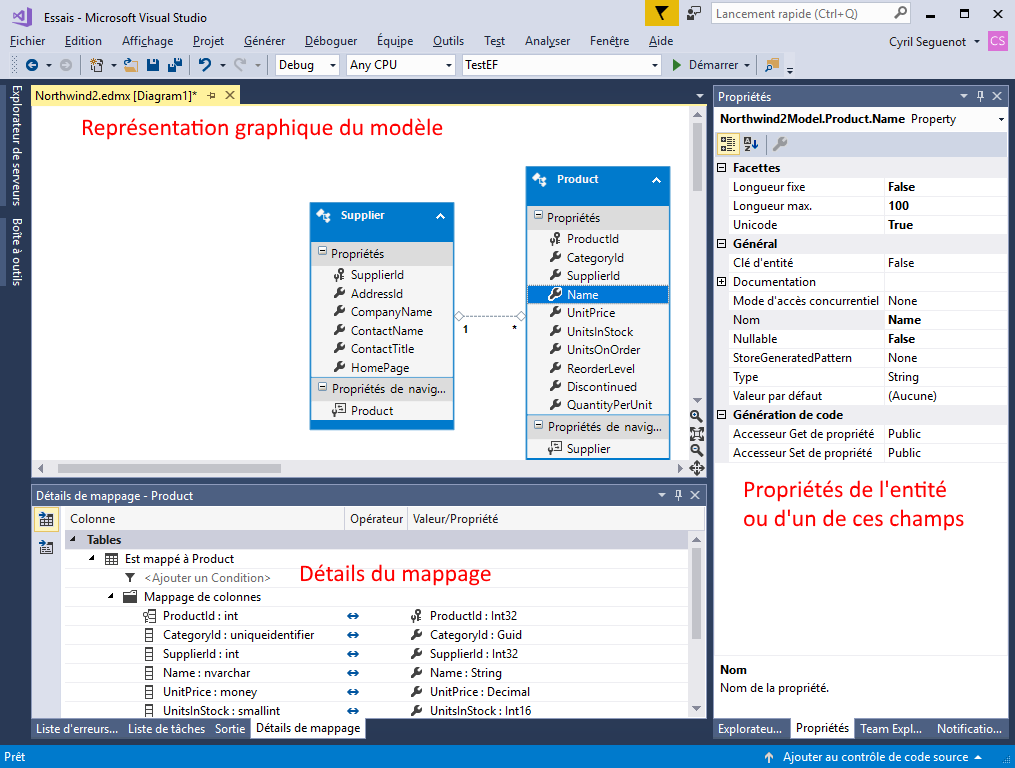
Ceci provoque l’ouverture de l’assistant EDM, grâce auquel on peut sélectionner :

* L’approche à mettre en œuvre (Code First, Model First, Database First)
* Le serveur et la base à utiliser si on a choisi une approche Database First ou Code First avec base existante
* La version d’Entity Framework à utiliser
* Les tables, vues et procédures stockées qu’on souhaite inclure dans le modèle
* Le nom du modèle



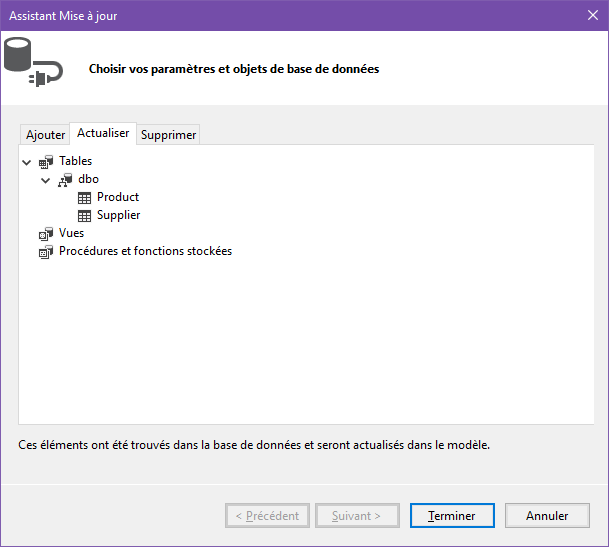
|  |  |
| --- | --- |
|  | L’assistant permet de générer les éléments suivants dans le projet :  Des références vers les assemblies :   * EntityFramework * EntityFramework.SqlServer, System.Runtime.Serialization * System.Security   Une chaîne de connexion dans le fichier App.config, avec un format spécial, car elle contient des informations relatives au fichier edmx  Un fichier .edmx représentant le modèle conceptuel, le modèle de stockage et le mappage entre les deux.  Une couche objet composée de classes générées à partir du modèle conceptuel (ces classes s’affichent sous le nœud du fichier edmx dans l’explorateur de solution) |

A la fin de l’assistant, le designer EDM s’ouvre et permet de modifier les propriétés du modèle et du mappage.



NB/ Les tables d’associations ne contenant que des champs de clés étrangères n’ont pas d’entités correspondantes dans le modèle conceptuel. Elles sont représentées par des relations plusieurs à plusieurs.

Si la structure de la base de données change, on peut mettre à jour le modèle conceptuel au moyen de l’assistant de mise à jour, qui peut être lancé à partir du menu contextuel « Mettre à jour le modèle à partir de la base de données » du modèle :



### Avec Entity Framework Core 2

**Ajout des références à EF Core**

Entity Framework Core peut être ajouté au projet en installant des packages NuGet. Pour cela, lancer la console en cliquant sur le menu « Outils \ Gestionnaire de package NuGet \ Console du gestionnaire de package », et saisir les commandes suivantes :

Install-Package Microsoft.EntityFrameworkCore.SqlServer  
Install-Package Microsoft.EntityFrameworkCore.SqlServer.Design  
Install-Package Microsoft.EntityFrameworkCore.Tools

Les packages sont installés dans le répertoire global de packages (par défaut C:\Users\xxxx\.nuget\packages), et référencés dans le projet sous le nœud « Dépendances \ NuGet »

Les deux premiers packages contiennent l’implémentation d’EF Core pour SQL Server (il existe d’autres fournisseurs de bases de données pour EF Core).

Le dernier contient des outils en ligne de commande pour générer et maintenir un modèle.

**Création du modèle**

EF Core ne génère pas de fichier modèle edmx, mais directement un jeu d'ntités POCO. Il décrit toutes les relations entre entités, et les contraintes avec l'API Fluent que nous verrons plus loin.

Pour générer ce modèle, on utilie l'utilitaire **Scaffold-DbContext**, installé avec le package NuGet Microsoft.EntityFrameworkCore.Tools

Voici un exemple de syntaxe de mise en oeuvre, avec un fournisseur SQL Server, et une sélection des tables à traiter.

Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Northwind2;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -Context Northwind2Context -OutputDir Models -Tables Employee, EmployeeTerritory, Territory, Region, Category, Product, Supplier, Address

La syntaxe complète de la commande Scaffold-DbContext est donnée sur [cette page](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/powershell) docs.microsoft.com.

* La partie entre doubles-côtes est la chaîne de connexion à la base de données. Elle peut être aussi au format suivant :  
  "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Northwind2;Integrated Security=True"
* Microsoft.EntityFrameworkCore.SqlServer est le fournisseur EF Core à utiliser (en l’occurence celui pour SQL Server)
* -OutputDir Models : spécifie que les entités POCO seront générées dans un sous-répertoire “Models” du projet
* -Tables Employee,... : spécifie la liste des tables de la base pour lesquelles ont souhaite créer des entités. Si on ne met pas ce paramètre, toutes les tables sont traitées.

Après exécution de cette commande, on peut voir le résultat suivant dans l’explorateur de solutions :

|  |  |
| --- | --- |
|  | On a une entité POCO par table, ainsi qu’une classe Northwind2Context, dérivée de DbContext, qui constitue le point d’accès central au modèle. |

## Création du modèle dans l’approche Code First

Dans cette approche, on crée soi-même un modèle avec des classes d’entités et des relations entre elles.

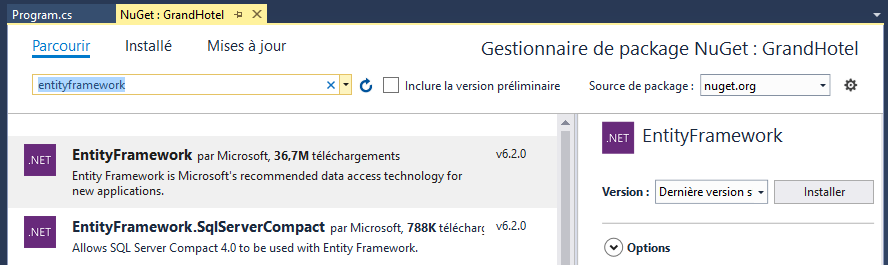
Si la base de données existe déjà, le modèle créé doit être en phase avec les tables de la base pour que le mappage soit possible.

Si la base de données n’existe pas, on peut la générer à partir du modèle.

### Ajout des références à EF

Comme précédemment, il faut commencer par ajouter les références aux dll d’EF en installant les packages NuGet correspondants. On peut le faire en ligne de commande ou avec l’interface graphique du gestionnaire de packages. Voici un exemple d’installation de EF 6 avec l’interface graphique :

Dans le menu contextuel du nœud « Références » du projet Visual Studio, sélectionner « Gérer les packages NuGet ».



Dans la fenêtre qui s’ouvre, cliquer sur l’onglet Parcourir, rechercher « EntityFramework », et l’installer. Ceci a pour effet de rajouter deux références au projet, ainsi que des entrées dans le fichier App.Config pour la gestion de la connexion.

### Création de la chaîne de connexion

Dans le fichier App.Config, ajouter une chaîne de connexion vers la base souhaitée, juste avant l’élément <EntityFramework>. Exemple :

<connectionStrings>

   <add name="BlogConnect"

        connectionString="Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Blog;Integrated Security=True"

        providerName="System.Data.SqlClient" />

</connectionStrings>

### Création des entités et relations

Créer des classes POCO correspondant aux tables de la base qu’on souhaite utiliser :

public class Classe

{

   public string Id { get; set; }

   // Propriété de navigation

   public virtual List<Etudiant> Etudiants { get; set; }

}

public class Etudiant

{

   public int Id { get; set; }

   // Clé étrangère

   public string ClasseId { get; set; }

   public string Nom { get; set; }

   public string Prenom { get; set; }

   // Propriété de navigation

   public virtual Classe Classe { get; set; }

}

Ces classes modélisent deux tables en relation 1,1 – 0,n (une classe est composée de plusieurs étudiants). La relation est modélisée dans le code au moyen des propriétés identifiants et des **propriétés de navigation**.

Si on veut avoir la possibilité de charger des données de détail de façon tardive, les propriétés de navigation doivent être virtuelles.

Pour que EF soit capable de mapper automatiquement les classes et propriétés sur les tables et champs de la base, et générer des requêtes SQL correctes à partir des requêtes Linq to Entities, il faut respecter certaines conventions (Cf. [cette page MSDN](https://msdn.microsoft.com/en-us/library/jj679962(v=vs.113).aspx)) :

**Conventions sur les noms de classes et propriétés**

* Les noms des entités doivent être identiques à ceux des tables, sans le pluriel. EF attend en effet des noms de table au pluriel, mais on peut supprimer cette convention (cf. plus loin)
* Les noms de propriétés doivent être identiques à ceux des champs des tables

**Conventions sur les clés primaires (PK) et clés étrangères (FK)**

* Les propriétés des entités du modèle qui sont nommées **Id** ou **<NomEntité>Id** sont reconnues comme clés primaires par EF.
* Les propriétés PK de type numérique sont générées comme des colonnes auto-incrémentées
* Une propriété de même type que la clé primaire d'une autre entité, et avec un nom qui obéit à un des modèles ci-dessous, est considérée comme clé étrangère :
  + <Nom de la prop de navigation><nom de la prop PK principale>
  + <Nom de la classe principale><nom de la PK>
  + <Nom de la PK de la classe principale>
* NB/ La classe principale est celle avec la cardinalité 1,1 dans une relation classique 1,1 - 0,n
* Si plusieurs propriétés correspondent à ces conditions, la priorité est donnée dans l'ordre ci-dessus.
* La détection de FK n'est pas sensible à la casse
* Le caractère nullable de la propriété FK détermine si elle est obligatoire ou non dans la relation
* Si une FK n'est pas nullable, la contrainte de suppression en cascade est appliquée. On peut modifier ce comportement via l'API Fluent.

Ainsi dans l’exemple précédent :

* Les propriétés Id des deux classes sont reconnues comme clés primaires
* La propriété ClasseId est reconnue comme clé étrangère, car elle obéit au modèle <Nom de la classe principale><nom de la PK>. Comme elle n'est pas nullable, tout étudiant est obligatoirement relié à une classe.
* La propriété de navigation Classe sera remplie avec la classe dont l’id vaut la valeur de ClasseId

Le respect des conventions décrites précédemment peut être contraignant et limitant. C’est pourquoi il est possible de personnaliser le modèle de données au moyen d’attributs sur les classes ou propriétés, et/ou au moyen de l’API Fluent, comme nous allons le voir dans les paragraphes qui suivent.

### Création du contexte de données

Pour exploiter le modèle de données, il faut créer une classe de contexte dérivée de DbContext. Elle doit contenir :

* Des propriétés de type DbSet correspondant aux différentes entités
* Un constructeur permettant de spécifier la chaîne de connexion à la base

public class BloggingContext : DbContext  
{  
 public DbSet<Blog> Blogs { get; set; }  
 public DbSet<Post> Posts { get; set; }  
  
 public BloggingContext() : base("name=EFCodeFirstConnect") { }  
}

Remarques :

* La récupération des données avec un DbSet ne fonctionne que si l’entité possède une clé primaire.
* DbContext est déclarée dans l’espace de noms System.Data.Entity pour EF6, et Microsoft.EntityFrameworkCore pour Ef Core
* Avec EF Core, la chaîne de connexion à la base est passée au contexte par injestion de dépendance. Voici la syntaxe utilisée :

public BloggingContext(DbContextOptions<BloggingContext> options) : base(options) { }

### Création de la base

Avec EF 6, si la base de données n’existe pas, elle sera créée automatiquement lors de la première utilisation du contexte de données.

Avec EF Core, il faut utiliser l’utilitaire de migration depuis la console du gestionnaire de package NuGet. Si la solution contient plusieurs projets, attention à bien sélectionner le bon projet dans la liste déroulante de la console :



Les commandes à saisir sont les suivantes :

Add-Migration "Creation initiale"

Update-Database

La première a pour effet de créer une classe de migration dans le dossier « Migrations » du projet.

La seconde génère la base de données. La console affiche le code SQL exécuté pour cela.

**Modification du modèle**

Si on modifie les entités du modèle après avoir généré la base, on obtient une erreur à l’exécution. EF Core produit une erreur de type SqlException, tandis que EF6 affiche le message :

The model backing the 'ContexteEcole' context has changed since the database was created. Consider using Code First Migrations to update the database

En effet, EF crée dans la base une table dbo.\_migrationsHistory afin de mémoriser le mappage entre les tables et les entités. Si son contenu n’est plus à jour, on obtient une erreur.

Il y a deux scénarios possibles :

* Soit on met à jour (migre) les tables de la base pour qu’elles reflètent les modifications du modèle défini par le code
* Soit on supprime l’historique de migration

La mise à jour des tables de la base à partir du modèle EF est un vaste sujet que nous ne traiterons pas ici. Pour supprimer l’historique de migration, il faut supprimer la table dbo.\_migrationsHistory dans la base, ainsi que le dossier « Migrations » du projet VS. On se retrouve alors dans une approche Code First avec base de données existante.

## Personnalisation du modèle avec des attributs

Les attributs de l’espace de noms System.ComponentModel.DataAnnotations.Schema décrivent le mappage des classes et propriétés sur les tables et champs correspondants dans la base de données. Voici la liste exhaustive :

|  |  |
| --- | --- |
| **Attribut** | **Description** |
| [Column](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.schema.columnattribute) | Nom de la colonne de table sur laquelle est mappée la propriété |
| [ComplexType](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.schema.complextypeattribute) | Marque l’entité comme étant un type complexe |
| [DatabaseGenerated](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.schema.databasegeneratedattribute) | Spécifie comment la base de données génère les valeurs de la propriété |
| [ForeignKey](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.schema.foreignkeyattribute) | Marque une propriété comme clé étrangère |
| [InverseProperty](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.schema.inversepropertyattribute) | Spécifie l’inverse d’une propriété de navigation |
| [NotMapped](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.schema.notmappedattribute) | Propriété non mappée avec une champ de table |
| [Table](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.schema.tableattribute) | Nom de la table sur laquelle est mappée la propriété |

Les attributs de l’espace de noms System.ComponentModel.DataAnnotations décrivent des contraintes de types, de valeurs, de format d’affichage et de validation sur les données. Voici les plus courants :

|  |  |
| --- | --- |
| **Attribut** | **Description** |
| [Display](https://msdn.microsoft.com/fr-fr/library/system.componentmodel.dataannotations.displayattribute(v=vs.110).aspx) | Permet de spécifier divers libellés associés à une entité ou propriété |
| [Key](https://msdn.microsoft.com/fr-fr/library/system.componentmodel.dataannotations.keyattribute(v=vs.110).aspx) | Marque une propriété comme clé primaire |
| [MaxLength](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.maxlengthattribute) | Longueur maximale pour la valeur de la propriété |
| [Required](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.requiredattribute) | Marque la propriété comme obligatoire |
| [StringLength](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.stringlengthattribute) | Longueur pour la valeur de la propriété |
| [Timestamp](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.timestampattribute) | Marque une propriété comme étant un intervalle de temps |
| ... |  |

Exemple :

public class Etudiant

{

   [Key]

   public int IdEtudiant { get; set; }

   [ForeignKey("Classe")]

   public string IdClasse { get; set; }

   public string Nom { get; set; }

   public string Prenom { get; set; }

   // Propriétés de navigation

   public virtual Classe Classe { get; set; }

   public virtual Adresse Adresse { get; set; }

}

public class Adresse

{

   [Key]

   [ForeignKey("Etud")]

   public int IdEtudiant { get; set; }

   public string Rue { get; set; }

   [Display(ShortName="CP")]

   [MaxLength(5)]

   public string CodePostal { get; set; }

   public string Ville { get; set; }

   // Propriété de navigation

   public virtual Etudiant Etud { get; set; }

}

Dans cet exemple, un étudiant est lié à une seule adresse. La propriété IdEtudiant est à la fois clé étrangère et clé primaire dans l’entité Adresse. Comme les propriétés ne respectent pas les conventions décrites plus haut, on utilise des attributs Key et ForeignKey pour marquer les clés primaires et étrangères.

L’attribut ForeignKey doit préciser le nom de la propriété de navigation qui représente l’entité principale.

L’attribut Display indique que la propriété CodePostal devra être affichée avec le libellé court « CP » dans certains composants d’affichage tels que la grille.

## Personnalisation du modèle avec l’API Fluent

L’API[Fluent](https://msdn.microsoft.com/en-us/library/jj591617(v=vs.113).aspx) offre des possibilités supplémentaires par rapport aux attributs. Elle permet notamment de :

* Ajouter ou supprimer certaines conventions
* Spécifier des noms de tables et colonnes différents de ceux des entités
* Spécifier des noms de schémas
* Mapper une même entité vers plusieurs tables…etc.
* …

NB/ Pour définir ses propres conventions, cf. [cette page MSDN](https://msdn.microsoft.com/en-us/library/jj819164(v=vs.113).aspx). Les attributs et l’API Fluent peuvent être utilisés conjointement.

Pour utiliser Fluent, il faut redéfinir la méthode OnModelCreating du DbContext, et utiliser la classe DbModelBuilder (EF 6) ou ModelBuilder (EF Core) :

using System.Data.Entity.ModelConfiguration.Conventions;  
  
protected override void OnModelCreating(DbModelBuilder modelBuilder)  
{  
 modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();  
 modelBuilder.Entity<Post>().Property(o => o.Rate).HasPrecision(2, 1);  
}

Dans cet exemple :

* On supprime la convention qui utilise des noms de tables au pluriel.
* On ajoute une convention qui définit le format décimal de la propriété Post.Rate

[ModelBuilder](https://docs.microsoft.com/en-us/ef/core/api/microsoft.entityframeworkcore.modelbuilder) est la classe qui construit le modèle de données interne du contexte. Elle est notamment utilisée pour accéder aux entités.

Les tables et colonnes de la base de données sont associées à des contraintes de clés primaires, clés étrangères, colonnes obligatoires…etc. Voici un exemple de retranscription de ces contraintes avec les méthodes de l’API Fluent, pour la base Northwind2 :

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

...

    modelBuilder.Entity<Employee>(entity =>

    {

        entity.Property(e => e.BirthDate).HasColumnType("date");

        entity.Property(e => e.FirstName)

            .IsRequired()

            .HasMaxLength(40);

        entity.Property(e => e.HireDate).HasColumnType("date");

        entity.Property(e => e.LastName)

            .IsRequired()

            .HasMaxLength(40);

        entity.Property(e => e.Notes).HasMaxLength(1000);

        entity.Property(e => e.Photo).HasColumnType("image");

        entity.Property(e => e.Title).HasMaxLength(40);

        entity.Property(e => e.TitleOfCourtesy).HasMaxLength(40);

        entity.HasOne(d => d.Address)

            .WithMany(p => p.Employee)

            .HasForeignKey(d => d.AddressId)

            .OnDelete(DeleteBehavior.Restrict)

            .HasConstraintName("Employee\_Address\_FK");

        entity.HasOne(d => d.ReportsToNavigation)

            .WithMany(p => p.InverseReportsToNavigation)

            .HasForeignKey(d => d.ReportsTo)

            .HasConstraintName("Employee\_Employee\_FK");

    });

...

OnModelCreating est appelée lorsque le modèle a été initialisé, mais avant qu’il ne soit utilisé pour initialiser le contexte. Ainsi, lorsqu’on utilise le contexte, les entités contenues dans ses DbSet contiennent bien les contraintes**.** Voici quelques explications sur les méthodes de l’API Fluent:

[PropertyBuilder<T>](https://docs.microsoft.com/en-us/ef/core/api/microsoft.entityframeworkcore.metadata.builders.propertybuilder-1) fournit des méthodes de configuration des propriétés des entités :

* IsRequired définit une propriété comme obligatoire
* HasMaxLength définit sa longueur maximale (pour une chaîne)
* HasColumnsType permet de préciser le type de la colonne correspondante dans la base

NB/ Les appels de méthodes peuvent être chaînés, car chaque méthode renvoie un objet de type PropertyBuilder.

[EntityTypeBuilder<T>](https://docs.microsoft.com/en-us/ef/core/api/microsoft.entityframeworkcore.metadata.builders.entitytypebuilder-1) fournit des méthodes de configuration des entités :

* HasOne permet d’établir une relation 1-1 avec une autre entité. En l’occurrence, un employé est en relation avec une adresse et un manager (propriété ReportsTo)
* HasMany permet d’établir une relation 1-n avec une autre entité. En l’occurrence, un employé est en relation avec plusieurs autres employés, si c’est un manager.

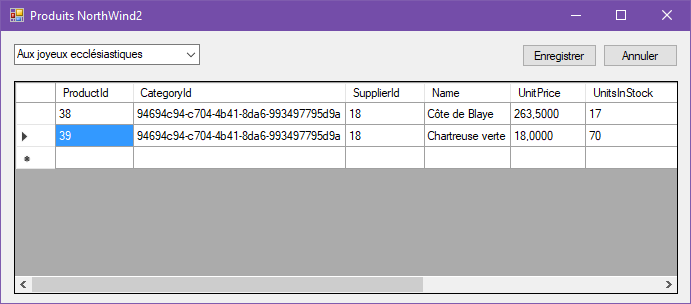
[ReferenceCollectionBuilder<T>](https://docs.microsoft.com/en-us/ef/core/api/microsoft.entityframeworkcore.metadata.builders.referencecollectionbuilder-2) fournit des méthodes de configuration des relations 1-n :

* HasForeignKey configure une propriété comme clé étrangère
* HasConstraintName est une méthode d’extension définie par la classe [RelationalReferenceCollectionBuilder](https://docs.microsoft.com/en-us/ef/core/api/microsoft.entityframeworkcore.relationalreferencecollectionbuilderextensions). Elle permet de spécifier le nom de la contrainte de clé étrangère.

Pour avoir une vue plus exhaustive de l’API Fluent pour EF Core, on pourra consulter les liens hypertextes fournis sur les noms de classes ci-dessus, ainsi que le site [Learn Entity Framework Core](http://www.learnentityframeworkcore.com/configuration/fluent-api).

## Chargement de données

Nous allons voir maintenant comment créer des requêtes LINQ sur la couche objet pour récupérer des données et les afficher. Notre objectif sera de créer une petite application Winforms qui affiche une liste déroulante de fournisseurs, et une grille détaillant les produits fournis par le fournisseur sélectionné :



NB/ Ce type d’application peut être réalisé de différentes façons. La plus rapide consiste à utiliser le composant BindingSource de Winforms, lié à une source de données de type objet. Cependant, notre but étant avant tout de découvrir les possibilités d’Entity Framework, je présente ci-dessous une solution plus « manuelle », mais aussi plus didactique.

using System;

using System.Data;

using System.Linq;

using System.Windows.Forms;

namespace DemoEF

{

   public partial class Form1 : Form

   {

      private Northwind2Entities \_contexte;

      public Form1()

      {

         InitializeComponent();

         cbFournisseurs.SelectedIndexChanged += CbFournisseurs\_SelectedIndexChanged;

         btnEnregistrer.Click += BtnEnregistrer\_Click;

      }

      // Au chargement du formulaire

      protected override void OnLoad(EventArgs e)

      {

         // Création du contexte

         \_contexte = new Northwind2Entities();

         // Requête LINQ pour récupérer les fournisseurs et leurs produits

var fournisseurs = \_contexte.Suppliers.Include(s => s.Products)  
 .OrderBy(s => s.CompanyName);

         // Définition de la source de données de la liste des fournisseurs

         cbFournisseurs.DataSource = fournisseurs.ToList();  
 cbFournisseurs.DisplayMember = "CompanyName";

      }

      // Au changement de sélection du fournisseur

      private void CbFournisseurs\_SelectedIndexChanged(object sender, EventArgs e)

      {

         // Affectation de la liste des produits du fournisseur sélectionné

         // comme source de données de la grille

         Supplier fourni = (Supplier)cbFournisseurs.SelectedItem;

         dgvProduits.DataSource = fourni.Products;

         // Masquage de certaines colonnes et retaillage

         dgvProduits.Columns["Supplier"].Visible = false;

         dgvProduits.AutoResizeColumns(DataGridViewAutoSizeColumnsMode.AllCells);

      }

      // A la fermeture du formulaire

      protected override void OnClosed(EventArgs e)

      {

         // Destruction du contexte  
 \_contexte.Dispose();

         base.OnClosed(e);

      }

   }

}

Analysons ce code :

### Gestion du contexte

Northwind2Entities est une classe dérivée de DbContext, générée automatiquement par EF, et permettant d’interagir avec le modèle. Comme elle est utilisée à chaque fois qu’on manipule des données, on l’instancie au chargement du formulaire, et on la détruit à la fermeture du formulaire. Si on l’utilisait dans une seule méthode, on utiliserait l’instruction using :

using (DbContext ctx = new Northwind2Entities()) { ... }

Le contexte contient des propriétés de type DbSet pour chaque entité du modèle. Ici, il s’agit des propriétés Suppliers et Products.

### Chargement des données

On a utilisé une requête LINQ unique pour récupérer en une seule fois l’ensemble des fournisseurs et de leurs produits. Pour récupérer les produits associés à chaque fournisseur, on a utilisé la méthode **Include** du DbSet Suppliers. C’est ce que l’on appelle du chargement immédiat (eagerly loading), par opposition au chargement tardif (lazy loading), qui consiste à charger les entités en relation au fur et à mesure des besoins.

Si on voulait charger progressivement les produits à chaque sélection d’un fournisseur, il suffirait donc de ne pas appeler Include dans la requête LINQ :

var fournisseurs = \_contexte.Suppliers.OrderBy(s => s.CompanyName);

NB/ le chargement tardif permet de limiter le volume de données à récupérer en une seule fois, mais génère plus de requêtes. Il faut donc déterminer au départ le type de chargement le plus adapté au contexte de notre application, en fonction du volume de données, de leur fréquence d’utilisation et de la bande-passante dont on dispose pour récupérer les données.

**Méthode ThenInclude de EF Core**

Include permet de charger un premier niveau de détail en même temps que les entités principales. Avec EF Core, il est possible de charger plusieurs niveaux de détail avec la méthode ThenInclude.

Par exemple, pour charger les fournisseurs, avec leurs produits, ainsi que les catégories de ces produits (pour récupérer le nom), on utilisera la syntaxe suivante :

var fournis = \_contexte.Suppliers.Include(s => s.Products).ThenInclude(p => p.Category)

ThenInclude() suit donc toujours un appel à Include().

### Affichage et vue locale

Dans Windows Forms, les contrôles d’affichage de données, tels que la ComboBox et la DataGridView possèdent une propriété DataSource pour spécifier la source des données à afficher. On affecte donc cette propriété avec le résultat de la requête LINQ, qu’on a préalablement transformé en liste à l’aide de ToList().

Il est important de garder à l’esprit que la requête LINQ n’est réellement exécutée que lorsqu’on énumère son résultat, par exemple en faisant appel à ToList. Une autre façon de faire serait la suivante :

// Chargement préalable des données dans le contexte

\_contexte.Suppliers.Include("Products").OrderBy(s => s.CompanyName).Load();

cbFournisseurs.DataSource = \_contexte.Suppliers.Local;

La méthode **Load** permet de charger immédiatement les données en mémoire dans le DbSet.

La propriété **Local** du DbSet représente une vue locale du DbSet, sans les enregistrements supprimés. Elle est de type ObservableCollection<T>, et reste toujours synchronisée avec le DbSet, c’est-à-dire que si l’on ajoute ou si l’on modifiée des entités dans la vue, elles sont également ajoutées ou modifiées dans le DbSet, et inversement. C’est pourquoi cette vue peut être utilisée comme source de données d’un contrôle. C’est aussi un bon moyen pour déterminer si un DbSet contient des enregistrements, sans provoquer son chargement. Exemple :

if (\_contexte.Suppliers.Any()) ...

// Provoque le chargement du DbSet

if (\_contexte.Suppliers.Local.Any()) ...

// Ne provoque pas le chargement du DbSet

S’il y a des fournisseurs dans la base, la première condition sera toujours vraie, car le fait d’appeler Any() provoque le chargement du DbSet des fournisseurs. En revanche, la seconde condition n’est vraie que si des données ont déjà été chargées au préalable dans le DbSet.

## Ajout, modification et suppression de données

Quand on récupère des données avec un DbContext, les entités créées à partir de ces données sont stockées en mémoire (i.e. mises en cache) dans des DbSet. Ces derniers permettent d’ajouter, modifier et supprimer facilement des entités.

### Dans une application console

On appellera directement les méthodes Add, Remove et Find du DbSet, comme illustré ci-dessous :

Ajout d’une entité dans le DbSet :

// Ajout d'un fournisseur

\_contexte.Suppliers.Add(fourni);

Pour la modification et la suppression, il faut déjà rechercher l’entité au moyen de la méthode Find :

// Recherche et suppression d'un fournisseur

fourni = \_contexte.Suppliers.Find(fourni.SupplierId);

if (fourni != null)

   \_contexte.Suppliers.Remove(fourni);

### Avec une interface graphique

Dans une application avec une interface graphique de type ASP.net, WPF ou WinForms, certains composants visuels peuvent être branchés directement sur les DbSet, et les opérations sur ces derniers deviennent alors transparentes. Considérons l’exemple ci-dessous, issu d’une application WinForms :

// Chargement préalable des données dans le contexte

\_contexte.Suppliers.Include(s => s.Products).OrderBy(s => s.CompanyName).Load();

// Branchement du DbSet des fournisseurs sur la liste déroulante

cbFournisseurs.DataSource = \_contexte.Suppliers.Local;

cbFournisseurs.DisplayMember = "CompanyName";

Affichage et saisie de produits au travers d’une DataGridView :

// Affectation de la liste des produits du fournisseur sélectionné

// comme source de données de la DataGridView

Supplier fourni = (Supplier)cbFournisseurs.SelectedItem;

dgvProduits.DataSource = fourni.Products;

Dans cet exemple, le composant DataGridView permet d’ajouter des produits en cliquant sur la ligne vide, de les éditer en cliquant sur les cellules, et d’en supprimer en sélectionnant une ou plusieurs lignes, et en appuyant sur la touche Suppr.

Il est important de garder à l’esprit que les entités des DbSet peuvent être modifiées par l’interface utilisateur, mais aussi par certains traitements faits dans le code (par exemple, calcul automatique de l’Id). Pour que l’interface visuelle reflète bien l’état des entités, les propriétés de type collection doivent être des collections de type ObservableCollection, ou bien BindingList. La seconde possède en plus des fonctionnalités de tri et recherche.

Lorsque les entités sont générées par EF, par défaut les collections sont de type ICollection. Il faut donc modifier ce type. On peut le faire de façon centralisée pour toutes les entités, en éditant le fichier XXXModel.tt. Il s’agit d’un fichier modèle utilisé par EF pour générer les classes d’entités. Il faut remplacer :

* Les deux occurrences de ICollection, et la première occurrence de HashSet par BindingList
* System.Collections.Generic par System.ComponentModel

… puis enregistrer le fichier, ce qui a pour effet de régénérer les entités.

## Enregistrement des modifications

EF permet d’enregistrer dans la base les modifications faites sur les entités du contexte, grâce à la méthode SaveChanges de DbContext. Il n’est cependant pas capable de détecter les entités en relation qui ont été supprimées, c’est pourquoi on doit les marquer nous-mêmes comme supprimées, comme le montre l’exemple ci-dessous :

// Au clic sur le bouton Enregistrer

private void BtnEnregistrer\_Click(object sender, EventArgs e)

{

   // EF n'étant pas capable de détecter les entités en relation

   // qui ont été supprimées, il faut les supprimer soi-même

   // Ce sont celles qui n'ont plus de lien avec l'entité parente

   foreach (var p in \_contexte.Products.Local.ToList())

   {

      if (p.Supplier == null)

         \_contexte.Products.Remove(p);

   }

   \_contexte.SaveChanges();

}

Voici comment fonctionne l’enregistrement des modifications en interne :

Le DbContext assure un suivi de toutes les modifications réalisées sur les objets issus d’une requête LINQ. Il est donc ensuite capable de transmettre ces modifications à la base de données, via sa méthode SaveChanges, qui construit et exécute en interne une requête SQL de type INSERT, UPDATE ou DELETE.

On peut faire plusieurs modifications avant l’appel de la méthode SaveChanges. Celle-ci exécute alors toutes les mises à jour sur la base au sein d’une transaction créée par le DbContext. Si l’une des mises à jour échoue, la transaction est interrompue, toutes les modifications sont annulées dans la base, et la méthode SaveChanges lève une exception. Les modifications restent toutefois présentes dans les objets en mémoire.

## Annulation des modifications

Pour assurer le suivi des modifications des données, le DbContexct stocke en interne les valeurs initiales des entités (celles chargées initialement depuis la base). On peut donc annuler les modifications en cours, et restaurer les valeurs initiales, de la façon suivante :

// Au clic sur le bouton Annuler

private void BtnAnnuler\_Click(object sender, EventArgs e)

{

   foreach (var entry in \_contexte.ChangeTracker.Entries<Product>())

   {

      switch (entry.State)

      {

         case EntityState.Modified:

         case EntityState.Deleted:

            // On annule d'abord les modifications de l’entité supprimée

            entry.State = EntityState.Modified;

            entry.State = EntityState.Unchanged;

            break;

         case EntityState.Added:

            entry.State = EntityState.Detached;

            break;

      }

   }

}

La propriété ChangeTracker du DbContext permet d’avoir accès aux informations de suivi des modifications. Sa propriété Entries<T> est une collection de DbEntryEntity, qui donne accès aux états des entités suivies par le contexte. On peut cibler un type d’entité particulier, ou bien tous les types si on ne spécifie pas le paramètre de type <T>.

Une entité supprimée a éventuellement pu être auparavant modifiée. On la repasse donc d’abord à l’état Modified, puis on annule les modifications en passant l’entité à l’état Unchanged.

L’état Detached est l’état par défaut lorsqu’on crée une nouvelle entité. Il indique que l’entité ne fait pas l’objet d’un suivi des modifications (en effet, une entité nouvellement créée n’existe pas encore en base).

NB/ DbChangeTracker possède également une méthode **HasChanges**, qui renvoie True si des modifications ont été faites sur le contexte depuis le chargement des données. Ceci peut être utile pour demander à l’utilisateur s’il souhaite enregistrer les modifications en cours, avant de fermer une fenêtre de saisie par exemple.

## Exécution de requêtes SQL

EF permet aussi d’exécuter des requêtes en langage SQL (cf. [cette page MSDN](https://msdn.microsoft.com/en-us/library/jj592907(v=vs.113).aspx)), de plusieurs façons :

* A partir de la méthode **SqlQuery d’un DbSet**

var prods = \_contexte.Products.SqlQuery("select \* from Product where SupplierId = 18");

La requête doit être de type select et doit obligatoirement retourner des données compatibles avec le type d’entité du DbSet (des produits dans cet exemple). Son résultat fait l’objet d’un suivi par le DbContext.

* A partir de la méthode **SqlQuery de la classe Database**

decimal prixMoyen = \_contexte.Database.SqlQuery<decimal>(  
 "select AVG(UnitPrice) from Product").Single();

Cela permet d’exécuter une requête select sans suivi des modifications par le DbContext.

* A partir de la méthode **ExecuteSqlCommand** de la classe Database

\_contexte.Database.ExecuteSqlCommand(@"update Product set UnitPrice = UnitPrice \* 1.05  
 where SupplierId = 18");

Cela permet d’exécuter des requêtes de type insert, update ou delete.

* A partir d’une commande classique

DbConnection conn = \_contexte.Database.Connection;

using (DbCommand cmd = conn.CreateCommand())

{

   cmd.CommandText = @"update Product set UnitPrice = UnitPrice \* 1.05  
 where SupplierId = 18"";

   conn.Open();

   cmd.ExecuteNonQuery();

   conn.Close();

}

Le point le plus intéressant ici, est que ce code reste indépendant du type de base de données, puisqu'on utilise les types abstraits DbConnection et DbCommand. A l'exécution, si la base utilisée est une base SQL Server, ce sont des instances de SqlConnection et SqlCommand qui seront créées.

**Remarque :** dans EF Core, la propriété Database.Connection et la méthode DbSet.SqlQuery sont remplacées respectivement par les méthodes Database.GetDbConnection et DbSet.FromSql.

## Gestion des conflits

### Exemples de scénarios à conflit

**Scénario 1 :**

Dans l’exemple précédent, prenons le cas du produit N° 34 dont le prix unitaire est de 14 € et la quantité en stock de 111. Supposons que deux utilisateurs U1et U2 affichent ce produit et décident de le modifier. U1 passe le prix unitaire à 15€, tandis que U2 passe la quantité en stock à 150. U1 enregistre ses modifications, puis U2 les siennes.

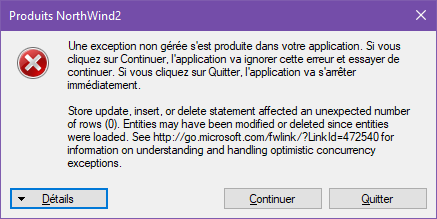
Quelles valeurs obtient-on en base ?

Par défaut, c’est le dernier qui a enregistré qui l’emporte : le produit garde donc un prix de 14 €, mais sa quantité en stock passe à 150. La modification faite par l’utilisateur U1 est donc perdue, sans qu’il le sache.

**Scénario 2 :**

U1 supprime le produit puis enregistre, tandis que U2 met à jour sa quantité, puis enregistre.

Dans ce cas, une exception de type DbUpdateConcurrencyException se produira lorsqu’U2 tentera d’enregistrer, et le message suivant s’affichera :



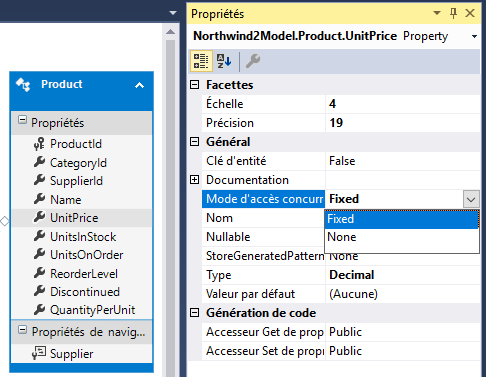
Si on estime que les conflits seront suffisamment rares, et si le comportement par défaut d’Entity Framework nous parait acceptable, il n’est pas nécessaire de mettre en place une gestion des conflits.

Dans le cas contraire, on peut aller plus ou moins loin dans la gestion. Voici quelques possibilités :

### Affichage d’un message explicite

Pour le scénario 1, on peut faire en sorte qu’une exception soit levée au moment où U2 enregistre ses modifications, de façon à l’informer du problème. Il sera ainsi géré de la même façon que le scénario 2.

Pour cela, dans le modèle, il faut modifier la propriété «**mode d’accès concurrentiel** » sur les propriétés pour lesquelles on souhaite gérer les conflits, comme le montre la figure ci-dessous :



Si on passe le mode d’accès concurrentiel à Fixed sur la propriété UnitPrice, EF conserve une copie de la valeur originale de cette propriété. Au moment de l’appel à SaveChanges, si la valeur en base est différente de la valeur originale, une exception DbUpdateConcurrencyException est levée, et la transaction est annulée. On peut alors intercepter l’exception et afficher un message d’erreur pour informer l’utilisateur, comme dans l’exemple suivant :

// Au clic sur le bouton Enregistrer

private void BtnEnregistrer\_Click(object sender, EventArgs e)

{

foreach (var c in \_contexte.Products.Local.ToList())

{

    if (c.Supplier == null)

       \_contexte.Products.Remove(c);

}

   try

   {

      contexte.SaveChanges();

   }

   catch (DbUpdateConcurrencyException ex)

   {

      // Construction d'un message d'avertissement

      StringBuilder sb = new StringBuilder();

      sb.AppendLine("Les prix des produits suivants ont été modifiés par d'autres utilisateurs :");

      // On récupère les id et nom des produits sur lesquels il y a un conflit

      foreach (DbEntityEntry entry in ex.Entries)

      {

         var p = (Product)entry.Entity;

         sb.AppendLine(p.ProductId + " - " + p.Name);

      }

      sb.AppendLine();

      sb.AppendLine("Rechargez les nouvelles valeurs et réappliquez vos modifications");

      MessageBox.Show(sb.ToString(), "Conflit", MessageBoxButtons.OK,

                        MessageBoxIcon.Warning);

   }

}

Dans cet exemple, on ne gère les conflits que sur un seul champ, en affichant un message explicite. Si on souhaite gérer les conflits sur l’ensemble des champs de l’entité, on aura avantage à utiliser la technique suivante :

* Dans la base de données, ajouter un champ de type [RowVersion](https://msdn.microsoft.com/fr-fr/library/ms182776(v=sql.120).aspx) (≡ Timestamp) dans la table correspondant à l’entité.
* Dans le modèle, affecter la propriété "Mode d'accès concurrentiel" à Fixed sur la propriété d’entité correspondant à ce champ.
* Dans le code, faire en sorte que la colonne Rowversion ne soit pas ajoutée à la grille, car cela provoque une erreur à l’affichage. Pour cela : dgvProduits.Columns.Remove("Rowversion");

RowVersion représente un nombre sur 8 octets généré automatiquement par le SGBD. Il est incrémenté à chaque opération d’insertion ou de mise à jour sur une table, quel que soit le ou les champs mis à jour. Le compteur est global à toute la base, et les valeurs générées sont uniques.

Si l’on utilise cette technique, on ne sait par contre pas sur quelle(s) propriété(s) porte le conflit. Il faut donc adapter le message d’erreur en conséquence. Dans notre cas, ce pourrait être par exemple :

"Les produits suivants ont été modifiés ou supprimés par d'autres utilisateurs :  
...  
Rechargez les nouvelles valeurs et réappliquez vos modifications"

### Gestion automatique des conflits

On peut gérer les conflits de façon automatique, sans alerter l’utilisateur, et sans perdre ses modifications, en utilisant la technique suivante :

using System.Data.Entity.Infrastructure;

...

// Autre alternative pour la gestion des conflits : gestion automatique

private void BtnEnregistrer\_Click(object sender, EventArgs e)

{

   foreach (var c in \_contexte.Products.Local.ToList())

   {

      if (c.Supplier == null)

         \_contexte.Products.Remove(c);

   }

   bool enregTerminé = false;

   while (!enregTerminé)

   {

      try

      {

         \_contexte.SaveChanges();

         enregTerminé = true;

      }

      catch (DbUpdateConcurrencyException ex)

      {

         // Récupère la ligne en erreur  
 DbEntityEntry ligneEnErreur = ex.Entries.Single();

         // Récupère les valeurs actuellement en base pour cette ligne

         DbPropertyValues valeursBDD = ligneEnErreur.GetDatabaseValues();

         if (valeursBDD == null)

         {

            // Si la ligne n'existe plus en base, on la supprime aussi localement

            //\_contexte.Products.Remove((Product)ligneEnErreur.Entity);

            ligneEnErreur.State = EntityState.Detached;

         }

         else

         {

            // Affecte ces valeurs comme valeurs d'origine de la ligne en erreur

            // tout en conservant les valeurs modifiées par l'utilisateur

            ligneEnErreur.OriginalValues.SetValues(valeursBDD);

         }

      }

   }

}

Pour comprendre ce code, il faut savoir que les lignes (qui représentent ici des produits) sont enregistrées en base une par une. Il ne peut donc y avoir qu’un seul conflit à la fois. On utilise une boucle while pour relancer l’enregistrement tant que tous les conflits n’ont pas été traités.

Pour gérer les conflits, on a besoin tout d’abord de récupérer les valeurs en base pour la ligne au moment de l’enregistrement.

Pour gérer le cas où la ligne a été supprimée par un autre utilisateur, on affecte l’état « inchangé » à la ligne. A la prochaine tentative d’enregistrement, la ligne ne sera ainsi plus prise en compte et la ligne ne sera plus visible dans la grille.

Pour gérer les autres cas, on fait en sorte que EF compare les valeurs saisies avec les valeurs actuelles en base, et non avec celles récupérées initialement à l’ouverture de la saisie. De cette façon, il n’y a plus de conflit, et les valeurs de la saisie sont enregistrées et l’emportent sur celles qui sont en base. On nomme cette approche « ClientWins »

Il existe d’autres façons de gérer les conflits (avec notamment une approche « Database wins »). Cf. [cette page](https://msdn.microsoft.com/en-us/data/jj592904) pour plus d’informations.

## Rafraîchissement des données

Dans un contexte multi-utilisateur, il peut être utile de fournir à l’utilisateur un moyen de recharger les données de la base pour récupérer les dernières mises à jour faites par les autres utilisateurs, sans avoir à fermer et rouvrir la fenêtre. Le moyen le plus simple est pour cela de recréer le contexte, et de relancer la requête initiale de chargement.

Si on ajoute un bouton Rafraîchir dans l’exemple précédent, on peut lui affecter le code suivant :

// Au clic sur le bouton Rafraîchir

private void BtnRafraichir\_Click(object sender, EventArgs e)

{

   \_contexte.Dispose();

   OnLoad(null);

}

## Cas particuliers

**Désactivation du suivi des modifications**

Le suivi des modifications réalisé sur une requête par le DbContext a évidemment un certain coût. Si on sait qu’on ne va pas modifier les données, on peut désactiver le suivi sur une requête au moyen de la méthode AsNoTracking de la classe DbSet. Ainsi, si on modifie la requête de l’exemple précédent de cette façon :

var fournisseurs = contexte.Suppliers.AsNoTracking().Include("Products").OrderBy(

                              s => s.CompanyName);

… alors les modifications qu’on pourrait réaliser sur les fournisseurs et leurs produits ne seront pas enregistrées en base lors d’un appel à SaveChanges.

NB/ Une autre syntaxe possible est la suivante :

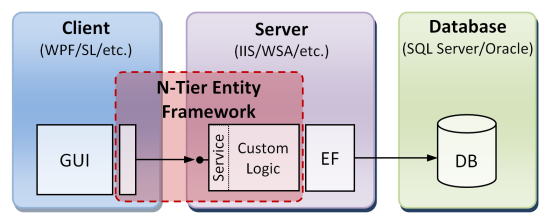
var fournisseurs = QueryableExtensions.AsNoTracking(

      contexte.Suppliers.Include("Products").OrderBy(s => s.CompanyName));

NB/ Ceci nécessite un using sur System.Data.Entity

**Suivi des modifications dans les applications N-Tiers**

Dans une applications N-Tiers (par exemple, avec couches client, service et BDD sur des machines différentes), les entités sont généralement déconnectées du contexte. Le suivi des modifications et le transfert au contexte nécessite alors la mise en place de solutions complémentaires telles qu’exposées sur [cette page MSDN](https://msdn.microsoft.com/en-us/library/jj613668(v=vs.113).aspx).



# Références bibliographiques

Documentation en ligne de [Microsoft Developer Network](https://msdn.microsoft.com/fr-fr/library/default.aspx) (MSDN)

Livre « [Visual C# 2010 Etape par étape](https://www.bookeenstore.com/ebook/9782100560943/visual-c-2010-etape-par-etape-john-sharp) » de John Sharp

Site [Entity Framework Tutorial](http://www.entityframeworktutorial.net)