

Secteur Tertiaire Informatique  
Filière « Etude et développement »

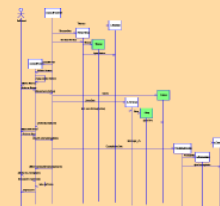
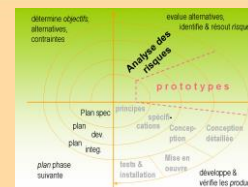
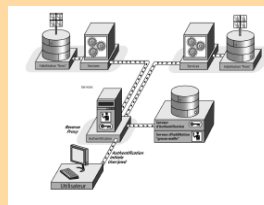
Construire une application organisée en couches en  
mettant en œuvre des Frameworks

**Mettre en œuvre un framework de persistance**  
**Manipuler les objets**

**Apprentissage**

Mise en situation

Evaluation





Version	Date	Auteur(s)	Action(s)
1.0	8/01/2020	Vincent Bost	Création du document

# TABLE DES MATIERES

Table des matières.....	4
1. Introduction.....	8
2. Installation du framework Entity Framework Core.....	8
3. Génération du modèle à partir d'une DB existante .....	9
3.1 Les paramètres de la commande Scaffold-DbContext .....	9
3.2 Le modèle des entités .....	10
4. Gestion des entités.....	11
4.1 Le contexte .....	11
4.2 Les états des entités .....	12
4.3 Le mode Connecté.....	13
4.4 Le mode déconnecté .....	14
4.4.1 Scénarios simples .....	16
4.4.2 Scénarios complexes .....	17
4.4.3 Conclusion sur le mode déconnecté.....	24
4.5 Mise à jour concurrentielle .....	24
4.6 Exécution différée ou immédiate .....	27
4.7 La projection .....	28
4.8 Récupérer un élément unique.....	29
4.9 Les imbrications .....	29
4.10 Les agrégations .....	30
4.10.1 Agrégation sur entités imbriquées .....	30
4.11 Les regroupements.....	30
4.12 Les jointures .....	33
4.12.1 Les jointures groupées .....	33
4.12.2 Les jointures simples.....	34
4.12.3 Remarques sur les jointures .....	35
4.13 Les partitions .....	37
5. Chargement immédiat ou différé .....	38
5.1 Le chargement différé (lazyLoading) .....	38
5.2 Le chargement immédiat (EagerLoading) .....	39
6. Arborescence d'expression IQueryable, IEnumerable .....	41
6.1 Arborescence d'Expression lambda.....	41

6.2	IQueryable et IEnumerable .....	41
6.3	Conclusion .....	43
7.	Exécuter des requêtes SQL brutes .....	43
7.1	Exécution de requêtes de sélection avec ou sans paramètres .....	43
8.	Opérateurs de requête standard.....	44

## Objectifs

Ce document a pour objet de vous accompagner dans l'apprentissage des techniques de manipulation d'objets au travers d'un Framework de persistance.

Après avoir réalisé les exercices proposés, vous devez être en mesure d'assurer

- De créer le modèle des entités à partir de la base de données et manipuler les objets la persistance de l'état de vos objets au sein d'une base de données

## Pré requis

Connaissance du langage C# et de l'environnement de développement Visual Studio

Connaissance du langage SQL.

Connaissance de la programmation orienté objet.

## Outils de développement

Environnement de développement :

- Visual Studio version 2019 ou ultérieure
- Le framework .Net Core 3.1

Framework de persistance :

- Entity Framework Core 3.1 ou ultérieure

Pour la manipulation des objets :

- Le langage de requête Linq

Support SGBDR pour persistance des données :

- SQL Server 2008 R2 ou ultérieure

## Méthodologie

Vous avez assimilé lors de la précédente phase d'apprentissage la notion d'ORM et les bases de l'élaboration de requêtes avec Linq.

Nous allons aborder ici la manipulation des objets issus de la couche de persistance.

## Mode d'emploi

Symboles utilisés :



Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.



Propose des exercices ou des mises en situation pratiques.



Point important qui mérite d'être souligné !

## Ressources

Script de création de la base de données AFPA

Script de création et d'initialisation de la base de données ComptoirAnglais\_

## Lectures conseillées

Le tutoriel en ligne consacré à Entity Framework(en) <http://www.entityframeworktutorial.net/>

Le tutoriel en ligne consacré à Linq (en) <http://www.tutorialsteacher.com/linq/linq-tutorials>

## 1. INTRODUCTION

Nous allons ici nous focaliser sur l'interrogation des données issues d'une base de données relationnelles SQL Server à l'aide du langage de requête Linq et de la cible Entity Core.

Les exercices sont à réalisés avec Visual Studio

Nous nous intéresserons plus particulièrement à la manipulation et la gestion de la persistance de nos objets.

Nous découvrirons comment :

- Ajouter un objet en base de données
- Modifier un objet
- Supprimer un objet

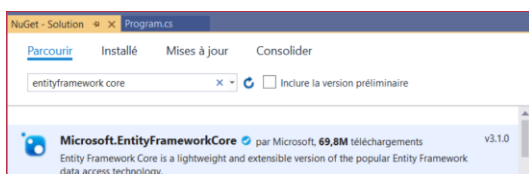
Mais aussi comment

- Extraire, Trier et Agréger nos objets.

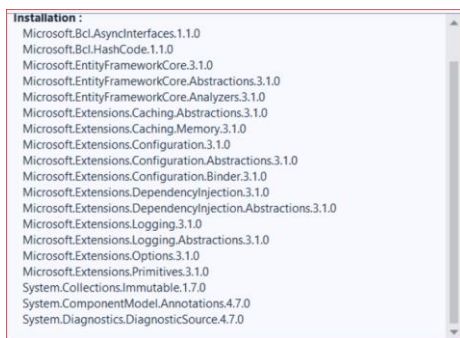
## 2. INSTALLATION DU FRAMEWORK ENTITY FRAMEWORK CORE

Créez un nouveau projet de type Application Console Core.

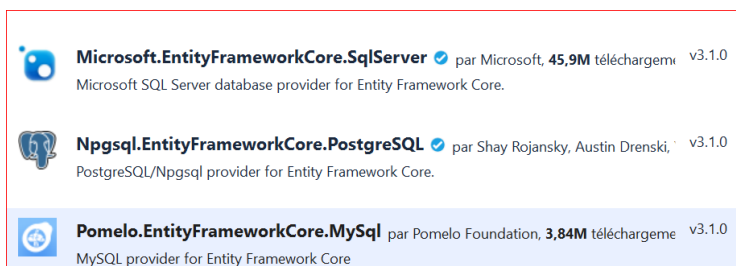
Ajoutez les packages nécessaires pour utiliser Entity Framework Core. A partir du gestionnaire de package Nuget :



Sont affichées les librairies dont dépend EF Core qui seront aussi installées.



Installez le fournisseur pour SqlServer. J'ai laissé dans la capture la liste d'autres fournisseurs pour les bases de données MySQL et PostgreSQL



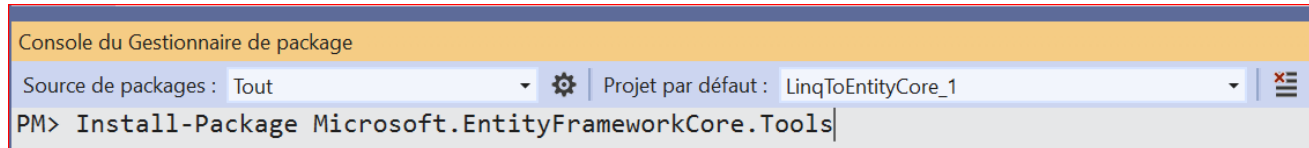
L'installation prendra en charge l'installation des bibliothèques nécessaires et manquantes.



### 3. GENERATION DU MODELE A PARTIR D'UNE DB EXISTANTE

Nous allons ici créer le contexte et les classes d'entités pour une base de données existante dans Entity Framework Core 3.1. La création de classes d'entités et de contexte pour une base de données existante est appelée mode **Database-First**.

Nous utiliserons pour cela les outils en ligne de commande qui nous sont fournis par un package Nuget . Lancer la console du Package Manager puis la commande suivante :



```
Console du Gestionnaire de package
Source de packages : Tout
Projet par défaut : LinqToEntityCore_1
PM> Install-Package Microsoft.EntityFrameworkCore.Tools
```

La commande de reverse engineering **Scaffold-DbContext** crée des classes d'entité et de contexte (en dérivant de la classe de base **DbContext**) en fonction du schéma de la base de données existante.

Créons les classes d'entité et de contexte pour la base de données ComptoirAnglais hébergée sur un server SQL.

#### 3.1 LES PARAMETRES DE LA COMMANDE SCAFFOLD-DBCONTEXT

Les paramètres suivants peuvent être spécifiés avec **Scaffold-DbContext** dans la console du gestionnaire de packages :

```
Scaffold-DbContext [-Connection] [-Provider] [-OutputDir] [-Context] [-Schemas>]
[-Tables>] [-DataAnnotations] [-Force] [-Project] [-StartupProject] [<CommonParameters>]
```

Dans Visual Studio, sélectionnez le menu Outils -> Gestionnaire de package NuGet -> Console du gestionnaire de package et exécutez la commande suivante qui précise une chaîne de connexion valide, le fournisseur de données et le répertoire dans lequel seront stockés les modèles d'entités.

```
PM>Scaffold-DbContext
"Server=localhost;Database=ComptoirAnglais_V1;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Context
ComptoirAnglaisEntities
```

Si vous souhaitez obtenir plus d'informations sur la commande

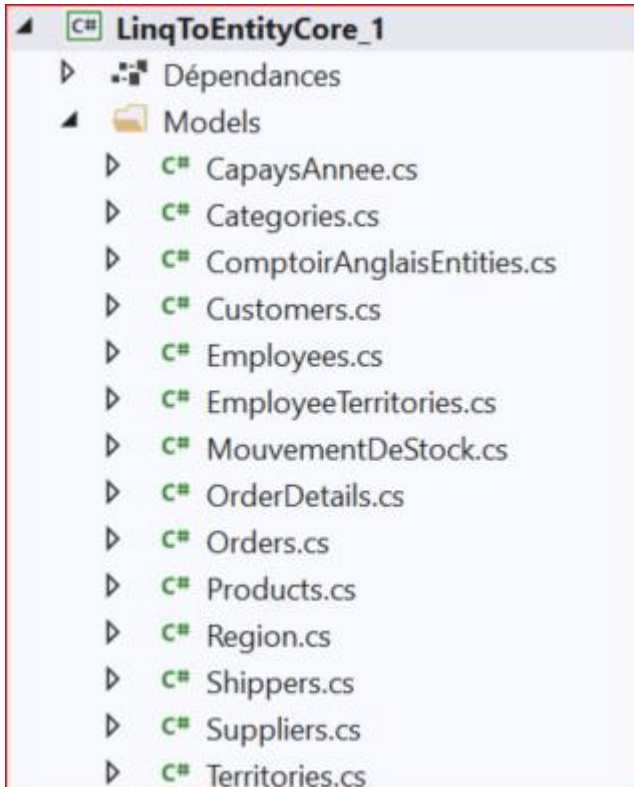
Utilisez la commande suivante pour obtenir l'aide détaillée sur la commande **Scaffold-DbContext**: `PM> get-help scaffold-dbcontext -detailed`

La commande **Scaffold-DbContext** ci-dessus a créé des classes d'entités pour chaque table de la base de données Comptoir Anglais de données et la classe de contexte (dérivé de **DbContext**) avec des configurations d'API Fluent pour toutes les entités.

Observons de plus près les objets créés.

### 3.2 LE MODELE DES ENTITES

L'outil à générer une version des modèles des entités POCO qui prend en compte l'ensemble des colonnes des tables et des vues présentes dans la base de données et la classe ComproirAnglaisEntities dérivé de DbContext qui vous sera présenté par la suite.



A noter : la présence des vues et c'est nouveau dans Entity Framework version core.

Vous pouvez observer la présence de propriétés de navigation issues de l'analyse des clés étrangères.

Zoom intéressant, celui de la table avec une clé étrangère qui référence cette même table.

Extrait du constructeur : Les propriétés de navigation qui permettent de cibler une référence multiple sont représentées par des HashSet.

Le principe est qu'au moment de la création de l'objet, les listes soient créées vides mais que les propriétés ne soient pas nulles.

```
0 références
public partial class Employees
{
    0 références
    public Employees()
    {
        EmployeeTerritories = new HashSet<EmployeeTerritories>();
        InverseReportsToNavigation = new HashSet<Employees>();
        Orders = new HashSet<Orders>();
    }
}
```

```
1 référence
public virtual Employees ReportsToNavigation { get; set; }
2 références
public virtual ICollection<EmployeeTerritories> EmployeeTerritories { get; set; }
2 références
public virtual ICollection<Employees> InverseReportsToNavigation { get; set; }
2 références
public virtual ICollection<Orders> Orders { get; set; }
```

## 4. GESTION DES ENTITES

### 4.1 LE CONTEXTE

Le contexte d'entités (DbContext) est une part importante d'Entity Framework.

Celui-ci constitue le pont entre les entités de votre domaine métier et les tables de la base de données où seront stockés les états de vos entités.

Il permet :

1. D'exposer les jeux d'entités qui correspondent aux tables de la base au travers des **DbSet<T>** où T représente un type d'entité.
2. D'assurer les requêtes auprès de la base de données.
3. D'assurer la conversion des données issues des requêtes SQL en objets.
4. D'assurer un suivi des modifications des entités
5. D'assurer la persistance des données.
6. De gérer les associations.
7. D'assurer la mise en cache des entités durant la durée de vie de ce dernier.

Le DbContext représente la **session** de la base de données.

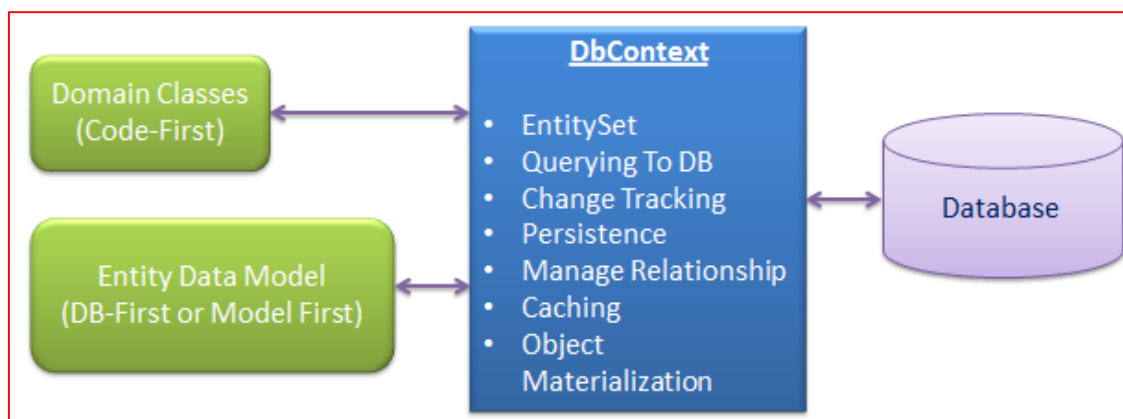


Figure 1 : Le contexte d'entités DbContext

💣 Toutes les **demandes de mise à jour** des entités en cache seront traitées comme **une seule transaction** lors de la demande de sauvegarde. Ces opérations seront considérées comme une seule unité de travail (Unit Of Work). La base de données demeure ainsi dans un état cohérent.

Observons un extrait du contexte d'entités.

Nous voyons les tables de la base de données exposées sous la forme d'un DbSet générique de type DbSet<T> où T est le type de l'entité.

Dans le cas d'une approche DataBase First ou Model First, les définitions des DbSet sont générées à partir du modèle.

Dans le cas d'une approche Code First, nous devons les définir.

L'instanciation du DbContext est réalisé à partir des informations stockées dans la chaîne de connexion dont le nom est passé au constructeur.

Si nous nous sommes basés sur un EDM pour le mappage des entités, la chaîne de connexion comporte les références à ce dernier.

## 4.2 LES ETATS DES ENTITES

Au cours de son cycle de vie, une entité peut connaître différents états.

Lors de la création et avant d'être attaché au contexte de données, elle sera dans un état **détaché**.

Pour que l'état d'une entité soit sauvegardé en base de données, elle devra être attachée au contexte de données et son état passera alors au stage **ajouté**.

Une entité existante pourra connaître les états :

- **Non modifié** : en absence de modification d'une valeur de ses propriétés
- **Modifié** : après modification de ses propriétés
- **Supprimé** : après demande de suppression

C'est l'état de l'entité qui déterminera la nature de la requête exécutée lors de la demande d'enregistrement des modifications en base de données.

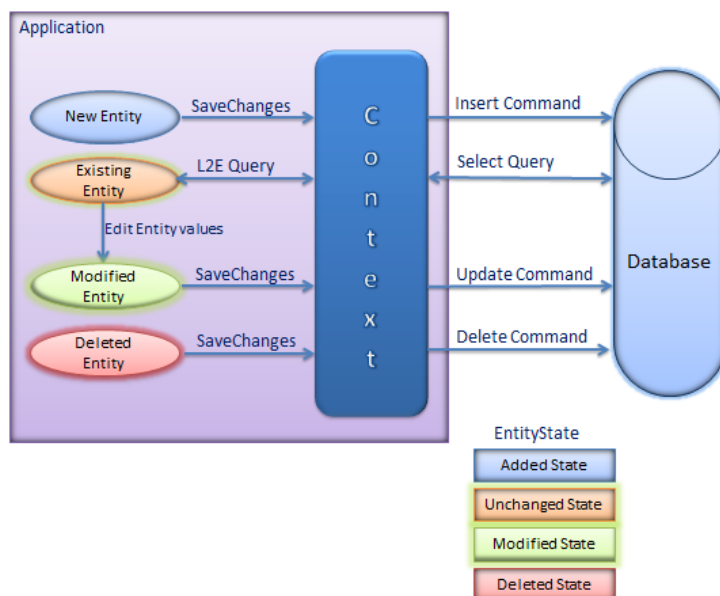


Figure 2 : Le cycle de vie d'une entité

Le type énuméré **EntityState** fournit la liste des états possibles.

Une fois que les commandes de mise à jour ont été exécutées par le DbContext sur invocation de sa méthode SaveChanges, l'état de l'entité sera non modifié, après opération de création ou de mise à jour, ou l'entité aura été supprimée du graphe dans le cas d'une demande de suppression.

### 4.3 LE MODE CONNECTE

Il est fondamental de distinguer les modes connecté et déconnecté d'Entity Framework.

Le choix de l'un ou l'autre de ces deux modes sera largement induit par la nature de l'application qui manipule les entités.

S'il s'agit d'une application lourde type Windows ou WPF, il est envisageable, mais pas nécessairement pertinent, d'opter pour le mode connecté. Le mode connecté doit être plutôt réservé à une architecture client/serveur à deux niveaux.

S'il s'agit d'une application de type Web ou Mobile, et plus généralement pour les applications sans état, il convient d'opter pour le mode déconnecté.

Les schémas présentés ci-après vont nous permettre de mieux appréhender pourquoi.

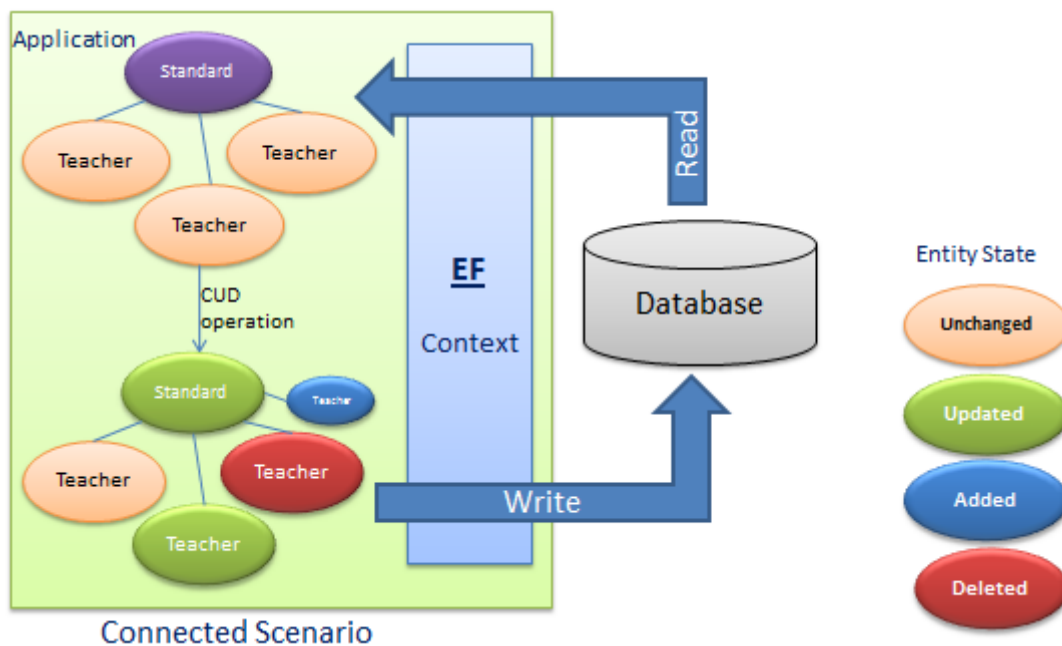


Figure 3 : Le scénario connecté

Dans un scénario connecté, le contexte de données (donc la session de base de données) doit être **maintenu tout au long du cycle**.

Dans ce scénario le **suivi des modifications (changetracker)** peut être assuré automatiquement par le contexte de données.

Observons le fonctionnement du tracking des modifications au travers d'une méthode qui extrait les propriétés des objets traqués et leurs états :



Lors de la demande de sauvegarde en base de données, le système déterminera donc à partir des éléments de tracking si une opération doit être réalisée ou non, et si oui, quelle opération effectuer.

L'ensemble des opérations seront considérées comme une seule unité de travail (une transaction globale). Ainsi, si une opération échoue, c'est l'ensemble des opérations qui seront rejetées.

#### 4.4 LE MODE DECONNECTE

Il est toujours plus pertinent et plus efficient dans le cadre des applications sans état de session. Dans ce scénario, pas de suivi automatique des modifications. Les entités doivent être ajoutées au graphe en précisant l'état de ces dernières.

Le contexte de données n'est pas maintenu et nous travaillons avec un nouveau contexte de données pour les opérations de sauvegarde.

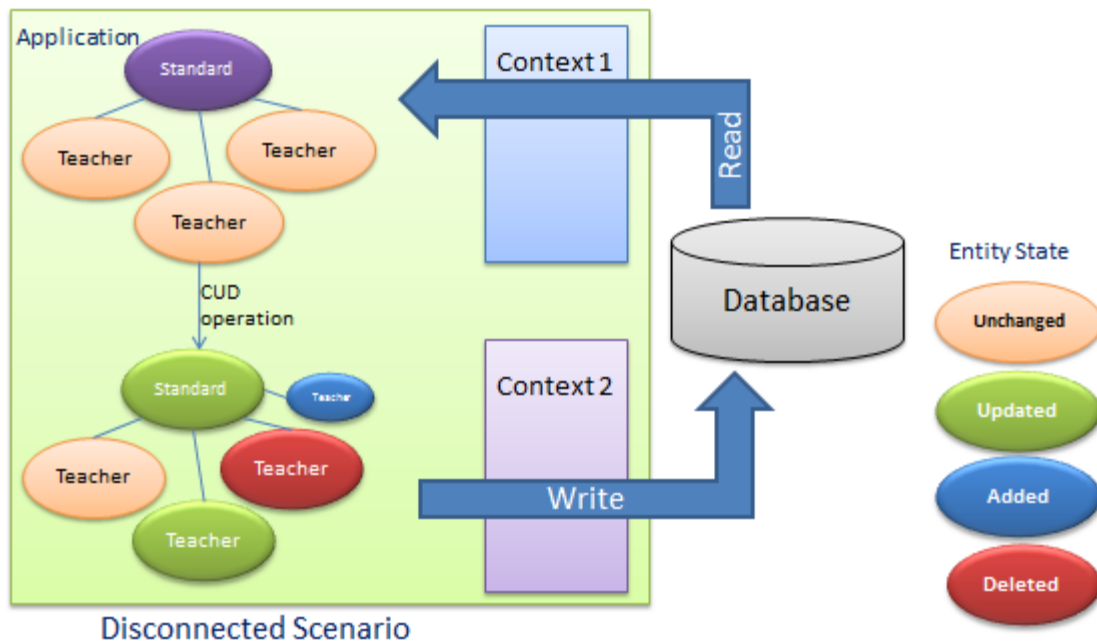


Figure 4 : Le scénario déconnecté

Imaginons le scénario classique de modification des informations d'un client à partir d'un formulaire Web :

1. Opération 1 : demande d'édition d'un client sélectionné dans une liste  
Requête GET avec identifiant Client
2. Opération 2 : composant serveur interroge la base de données et récupère l'entité client demandée. Une vue html est produite à partir des valeurs des propriétés de l'entité.
3. Opération 3 : demande d'enregistrement des modifications apportées via le formulaire.  
Requête POST avec collections de paramètres clés/valeurs
4. Opération 4 : composant serveur crée une nouvelle entité et alimente les propriétés de celle-ci à partir du dictionnaire clés/valeurs.  
L'entité est ajoutée avec l'état modifié.  
Demande de sauvegarde.

Il n'est pas envisageable de maintenir le contexte de données entre les opérations 2 et 4. Nous travaillons donc ici avec deux sessions de données différentes.

L'opération 4 nécessite ou non d'extraire les valeurs de l'entité depuis le serveur avant la modification. Si nous sommes en mesure de fournir toutes les valeurs des propriétés, alors cela ne sera pas utile. Sinon, nous devons charger l'état initial de l'entité préalablement à la modification des propriétés de celle-ci.

💣 Attention aux méthodes retenues pour l'ajout de l'entité dans le graphe comme illustré dans l'exemple page suivante. Une bonne pratique consiste à ajouter dans le graphe les entités parents comme les entités enfants en précisant systématiquement l'état de celles-ci.

Le schéma ci-dessous illustre la problématique. Ici, pas de détermination automatique de l'état de l'entité. Il faudra préciser celui-ci lors de l'ajout au contexte de données.

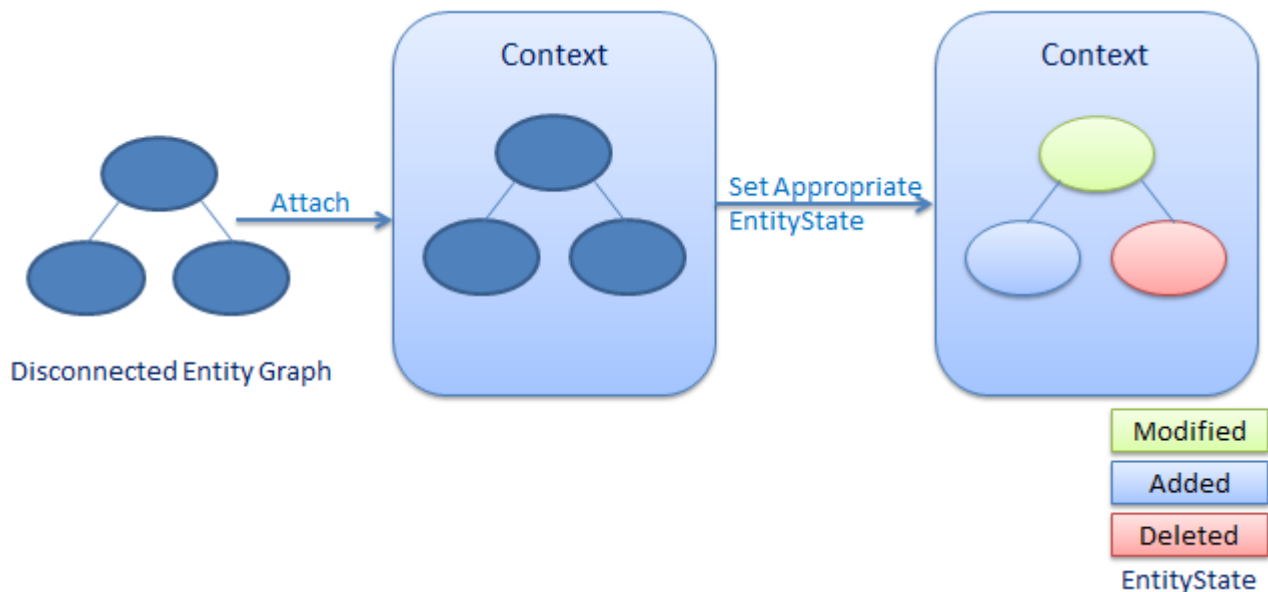


Figure 5 : Ajout d'un graphe d'entités au contexte

💣 Implémenter des scénarios de mise à jour des graphes d'entité en mode déconnecté est souvent nécessaire pour des besoins de performance accrue de nos applications. Mais nous allons pouvoir constater que leur **mise en œuvre est plus complexe**.

Le suivi automatique des changements d'état étant inopérant dans le mode déconnecté, nous modifions les propriétés du contexte de données pour ne pas effectuer celui-ci.

Nous allons aborder dans les pages à venir plusieurs scénarios de mise en œuvre :

1. Des scénarios simples mais limités le plus souvent à une transaction impliquant une seule entité.
2. Un scénario plus complexe mais assurément plus en mesure de satisfaire des contextes d'usage très diversifiés. Nous souhaitons prendre en compte l'ensemble des éléments présents dans le graphe objets en une seule transaction.

#### 4.4.1 Scénarios simples

##### 4.4.1.1 Ajout d'une seule entité

Ajout d'une entité client par la méthode Add du DbSet

Création d'un nouvel objet POCO de type Customers

```
Customers client = new Customers()
{
    CustomerID = "BOSTI",
    City = "Sainte Féréole",
    Address = "Le bois Colombes",
    CompanyName = "Boston Compagnie",
    ContactName = "Vincent Bost",
    IdPays2 = "FR"
};
```

Ajout de l'entité au contexte de données et demande de sauvegarde

```
using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Customers.Add(client);

    dbContext.Database.Log = c => TraceurEF.Tracer("Déconnecté Simple", c);
    dbContext.SaveChanges();
}
```

Une requête d'insertion paramétrée a été générée. Notre transaction a bien été enregistrée. Nous aurions pu aussi ajouter l'entité en précisant son état en recourant à la méthode Entry.

```
dbContext.Entry<Customers>(client).State = System.Data.Entity.EntityState.Added;
```

##### 4.4.1.2 Modification d'une entité

Dans ce scénario, il nous faut nous assurer que nous travaillons avec des entités déconnectées du contexte de données et sans proxys. Pour modifier une entité, dans le mode déconnecté sans suivi de modification, il convient de l'ajouter au contexte en précisant son état.

```
Customers client = null;
using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Configuration.ProxyCreationEnabled = false;
    client = dbContext.Customers.Find("BOSTI");
}
client.City = "Brive";

using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Entry(client).State = System.Data.Entity.EntityState.Modified;

    dbContext.Database.Log = c => TraceurEF.Tracer("Déconnecté Simple Modification entité", c);

    dbContext.SaveChanges();
}
```

Pas de difficultés ici pour la mise à jour d'une seule entité.



#### 4.4.1.3 Suppression d'une entité

La suppression d'une entité est assez similaire à la mise à jour si ce n'est que doivent être prises en considération les caractéristiques des associations avec les entités enfants.

Nous y reviendrons dans le chapitre consacré aux mécanismes de validation et exceptions.

```
Customers client = null;
using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Configuration.ProxyCreationEnabled = false;
    client = dbContext.Customers.Find("BOSTI");
}
using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Entry(client).State = System.Data.Entity.EntityState.Deleted;

    dbContext.Database.Log = c => TraceurEF.Tracer("Déconnecté Simple Suppression entité", c);

    dbContext.SaveChanges();
}
```

#### 4.4.2 Scénarios complexes

##### 4.4.2.1 Ajout d'un graphe d'entités - 1

Nous allons ici proposer un exemple d'ajout d'un graphe d'entités où chaque entité a été nouvellement créée. Il prend en compte la création d'un client associé à la création d'une commande.

L'ensemble des opérations doit être considéré comme une seule unité de travail. Ainsi, si une erreur survient au cours de cette transaction, aucune des opérations réalisées ne devra être validée.

Il n'est pas correct d'envisager que chaque entité fasse l'objet d'une transaction spécifique.

```
Customers client = new Customers()
{
    CustomerID = "BOSTI",
    City = "Sainte Féréole",
    Address = "Le bois Colombes",
    CompanyName = "Boston Compagnie",
    ContactName = "Vincent Bost",
    IdPays2 = "FR"
};
Orders commande = new Orders() { OrderDate = DateTime.Now };
commande.Order_Details.Add(new Order_Details()
{ ProductID = 2, UnitPrice = 15, Quantity = 3, Discount = 0 });
commande.Order_Details.Add(new Order_Details()
{ ProductID = 2, UnitPrice = 25, Quantity = 5, Discount = 0.05f });
client.Orders.Add(commande);

using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Entry<Customers>(client).State = System.Data.Entity.EntityState.Added;

    dbContext.Database.Log = c => TraceurEF.Tracer("Déconnecté Simple Ajout entités", c);

    dbContext.SaveChanges();
}
```

Trace du code SQL exécuté.

1. Début transaction
2. Ajout du client

```
Connexion ouverte à 05/03/2016 17:16:00 +01:00
: Déconnecté Simple Ajout entités
Début de la transaction à 05/03/2016 17:16:00 +01:00
: Déconnecté Simple Ajout entités
INSERT [dbo].[Customers]([CustomerID], [CompanyName], [ContactName],
VALUES (@0, @1, @2, NULL, @3, @4, NULL, NULL, NULL, NULL, @5)
: Déconnecté Simple Ajout entités
-- @0: 'BOSTI' (Type = AnsiStringFixedLength, Size = 5)
```

3. Ajout de la commande. Nous remarquons la lecture de l'enregistrement pour récupérer les valeurs générées côté serveur.

```
: Déconnecté Simple Ajout entités
INSERT [dbo].[Orders]([CustomerID], [EmployeeID], [Or
VALUES (@0, NULL, @1, NULL, NULL, NULL, NULL, NULL, N
SELECT [OrderID], [TS]
FROM [dbo].[Orders]
WHERE @@ROWCOUNT > 0 AND [OrderID] = scope_identity()
```

4. Ajout des lignes de commande. Nous remarquons là encore une lecture pour alimenter la valeur de la colonne de type Time Stamp (RowVersion) qui permettra de gérer les conflits de mises à jour concurrentielles. Voir le chapitre consacré à la validation.

```
: Déconnecté Simple Ajout entités
INSERT [dbo].[Order Details]([OrderID], [ProductID], [UnitPrice]
VALUES (@0, @1, @2, @3, @4)
SELECT [TS]
FROM [dbo].[Order Details]
WHERE @@ROWCOUNT > 0 AND [OrderID] = @0 AND [ProductID] = @1: I
```

L'ajout de la ligne suivante provoquera une erreur due à l'absence de la référence produit :

```

INSERT [dbo].[Order Details]([OrderID], [ProductID], [UnitPrice]
VALUES (@0, @1, @2, @3, @4)
SELECT [TS]
FROM [dbo].[Order Details]
WHERE @@ROWCOUNT > 0 AND [OrderID] = @0 AND [ProductID] = @1:

: Déconnecté Simple Ajout entités
-- @0: '11105' (Type = Int32)
: Déconnecté Simple Ajout entités
-- @1: '888' (Type = Int32)
: Déconnecté Simple Ajout entités
-- @2: '25' (Type = Decimal, Precision = 19, Scale = 4)
: Déconnecté Simple Ajout entités
-- @3: '5' (Type = Int16)
: Déconnecté Simple Ajout entités
-- @4: '0,05' (Type = Single)
: Déconnecté Simple Ajout entités
-- Exécution à 05/03/2016 17:16:01 +01:00
: Déconnecté Simple Ajout entités
-- Échec dans 118 ms avec l'erreur : L'instruction INSERT est e
L'instruction a été arrêtée.

```

Après vérification dans la base de données, nous pouvons constater qu'aucune opération n'a été enregistrée : cela confirme le fait que l'ensemble des opérations soit traité comme une seule transaction.

Nous voyons aussi que les opérations demandées sont conformes à notre attente.

Corrigeons l'erreur sur la référence produit et soumettons à nouveau notre demande.

Nous pouvons vérifier que les infos sont bien enregistrées dans la base de données.

	CustomerID	OrderID	Expr1	ProductID	UnitPrice	Quantity	Discount
►	BOSTI	11106	11106	2	15,0000	3	0
	BOSTI	11106	11106	3	25,0000	5	0,05

Mais que ce passe-t-il si nous avons fait des modifications sur un client ou des commandes existantes ?

Cette problématique est abordée au point suivant.

#### 4.4.2.2 Modification d'un graphe d'entités -2

Dans ce deuxième scénario, nous allons modifier la commande créée précédemment.

Nous indiquons que nous souhaitons obtenir les entités enfants immédiatement.

Nous reviendrons sur ce procédé dans le chapitre sur les chargements différé et immédiat.

```

Customers client = null;
using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Configuration.ProxyCreationEnabled = false;
    client = dbContext.Customers.Include("Orders.Order_Details").First(c =>c.CustomerID=="BOSTI");
}

```

Nom	Valeur
client	{LinqToEF_005.Customers}
Address	"Le bois Colombes"
City	"Sainte Féréole"
CompanyName	"Boston Compagnie"
ContactName	"Vincent Bost"
ContactTitle	null
Country	null
CustomerID	"BOSTI"
Fax	null
IdPays2	"FR"
Orders	Count = 1
Orders [0]	{LinqToEF_005.Orders}
CustomerID	"BOSTI"
Customers	{LinqToEF_005.Customers}
EmployeeID	null
Employees	null

Ajoutons une commande et modifions celle existante.

```
Orders commandeA = new Orders() { OrderDate = DateTime.Now, CustomerID=client.CustomerID };
commandeA.Order_Details.Add(new Order_Details()
{ ProductID = 3, UnitPrice = 25, Quantity = 5, Discount = 0.05f });
client.Orders.Add(commandeA);

Orders commandeM = client.Orders.First(o => o.OrderID == 11106);
commandeM.RequiredDate = DateTime.Now.AddDays(10);
```

Ajoutons le graphe au contexte de données et demandons à sauvegarder les modifications.

```
using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Entry<Customers>(client).State = System.Data.Entity.EntityState.Unchanged;

    dbContext.Database.Log = c => TraceurEF.Tracer("Modification Complexe", c);

    dbContext.SaveChanges();
}
```

Aucune modification n'est détectée et aucune sauvegarde des données modifiées ou ajoutées n'est réalisée.

Il ne s'agit donc pas de la bonne méthode...

#### 4.4.2.3 Modification d'un graphe d'entités -3

En fait, pour ajouter un graphe d'entités dont certaines ont pu être modifiées, ajoutées ou supprimées, à un contexte de données et assurer la sauvegarde des états de ces dernières, une seule solution que nous allons découvrir ici.

Il convient d'ajouter chaque entité dans le contexte de données en précisant si elle est ajoutée, modifiée, supprimée ou sans changement. En effet, nous n'avons, avec ce modèle déconnecté, aucune possibilité de détecter les changements d'état.

Cette technique va donc nécessiter de disposer, **au niveau de chaque type d'entité POCO**, d'une propriété reflétant **l'état de l'entité**.

Nous créons un type énuméré, que nous nommerons ici EntityPOCOState, des différents états possibles. Puis définissons une interface dont le contrat définit l'état de l'entité.

ORM Manipulation des objets

Cette interface doit être ensuite implémentée dans chaque type d'entité POCO.

**A noter :** Nous pourrions créer aussi une interface qui comprendrait l'ensemble des comportements à implémenter au sein d'une entité.

```
public enum EntityPOCOState
{
    Added,
    Deleted,
    Modified,
    Unchanged
}
3 références
public interface IEntityPOCOState
{
    10 références
    EntityPOCOState Etat
    {
        get;
        set;
    }
}
26 références
public partial class Customers : IEntityPOCOState
{
    private EntityPOCOState _etat = EntityPOCOState.Unchanged;
    10 références
    public EntityPOCOState Etat
    {
        get { return _etat; }
        set { _etat = value; }
    }
}
```

Après avoir implémenté l'interface au niveau de chaque type POCO du modèle, nous révisons le code de notre transaction afin de définir l'état qui résulte de chaque opération sur nos entités.

L'état par défaut est non modifié. Il s'agit notamment de l'état affecté lors de l'extraction des entités depuis la couche de persistance.

Extraction des données du client. Celui-ci et l'ensemble de ses entités associées, seront à l'état non modifié.

```
Customers client = null;
using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Configuration.ProxyCreationEnabled = false;
    client = dbContext.Customers.Include("Orders.Order_Details")
        .First(c => c.CustomerID == "BOSTI");
}
```

Modification du client :

```
client.ContactName = "Vincent";
client.Etat = EntityPOCOState.Modified;
```

Création d'une nouvelle commande :

```

Orders commandeA = new Orders()
{ OrderDate = DateTime.Now, CustomerID = client.CustomerID, Etat=EntityPOCOState.Added};

commandeA.Order_Details.Add(new Order_Details()
{ ProductID = 7, UnitPrice = 30, Quantity = 15,
  Discount = 0.05f,Etat=EntityPOCOState.Added });
client.Orders.Add(commandeA);

```

Modification d'une commande existante :

```

Orders commandeM = client.Orders.First(o => o.OrderID == 11106);
commandeM.RequiredDate = DateTime.Now.AddDays(10);
commandeM.Etat = EntityPOCOState.Modified;

```

Une fois nos opérations sur les entités du domaine terminées, il nous faut sauvegarder l'état de nos entités.

💣 Nous sommes dans l'obligation d'ajouter chaque entité au contexte de données en précisant son état. La classe représentant notre contexte de données a été modifiée et une méthode implémentée dont le seul rôle est de déterminer EntityState à partir d'EntityPOCOState. J'ai préféré cette solution à une concordance exacte entre EntityState et EntityPocoState qui reste envisageable.

```

using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Entry<Customers>(client).State = dbContext.DeterminerEtat(client.Etat);
    foreach (Orders cde in client.Orders)
    {
        dbContext.Entry<Orders>(cde).State = dbContext.DeterminerEtat(cde.Etat);

        foreach (Order_Details detail in cde.Order_Details)
        {
            dbContext.Entry(detail).State = dbContext.DeterminerEtat(detail.Etat);
        }
    }

    dbContext.Database.Log = c => TraceurEF.Tracer("Modification Complexe", c);

    dbContext.SaveChanges();
}

```

Méthode pour établir la correspondance entre les deux types énumérés.

```

public partial class ComptoirAnglaisEntities
{
    3 références
    public System.Data.Entity.EntityState DeterminerEtat(EntityPOCOState etat)
    {
        if (etat == EntityPOCOState.Added)
            return System.Data.Entity.EntityState.Added;
        if (etat == EntityPOCOState.Modified)
            return System.Data.Entity.EntityState.Modified;
        if (etat == EntityPOCOState.Deleted)
            return System.Data.Entity.EntityState.Deleted;
        else
            return System.Data.Entity.EntityState.Unchanged;
    }
}

```

Nous pouvons en lieu et place de la définition de cette méthode déclarer notre type énuméré avec les mêmes valeurs constantes que le type EntityState. Je n'ai pas retenu l'état détaché qui dans ce contexte est sans intérêt.

```
public enum EntityPOCOState
{
    // ...
    Unchanged = 2,
    // ...
    Added = 4,
    // ...
    Deleted = 8,
    // ...
    Modified = 16,
}
```

Et assurer la conversion d'un type énuméré à l'autre :

```
dbContext.Entry<Orders>(cde).State = (System.Data.Entity.EntityState)cde.Etat ;
```

Quelle que soit l'approche retenue, nous avons maintenant un résultat conforme à notre attente comme le montre l'affichage des éléments en base de données :

CustomerID	ContactName	OrderID	Expr1	ProductID	UnitPrice	Quantity	Discount	RequiredDate
BOSTI	Vincent	11106	11106	2	15,0000	3	0	2016-03-16 0...
BOSTI	Vincent	11106	11106	3	25,0000	5	0,05	2016-03-16 0...
BOSTI	Vincent	11107	11107	7	30,0000	15	0,05	NULL

#### 4.4.3 Conclusion sur le mode déconnecté

Si les opérations de mise à jour d'une entité unique sont simples à mettre en œuvre, la prise en charge de transactions complexes permettant de traiter un ensemble d'entités comme une unité de travail unique requiert des efforts supplémentaires notamment au niveau de la conception de nos classes du domaine métier.

Dans le mode déconnecté, le suivi automatique des modifications est sans effet. Si nous ajoutons à cela que les proxys, dans le cadre de scénarios d'architectures distribuées, posent des problèmes de sérialisation et de performances, la meilleure solution consiste à s'en passer.

Il est important, pour conserver de bonnes performances à nos applications, de maîtriser le moment où devons être chargées les entités associées.

Je conseille donc d'interdire le chargement différé des entités afin de déterminer au mieux le moment où elles devront être chargées.

#### 4.5 MISE A JOUR CONCURRENTIELLE

Dans un environnement multi-utilisateur, nous avons deux modèles de mise à jour des données d'une base de données :

- l'accès simultané optimiste : ce modèle ne prévoit pas de verrouillage des lignes de données pour mise à jour. C'est celui qui est privilégié aujourd'hui pour toutes les applications pouvant détenir les informations un temps assez long.
- l'accès simultané pessimiste lui nécessite le verrouillage des données : il reste surtout illustré par quelques activités particulières qui nécessitent un accès exclusif à l'information. La durée du verrouillage doit être la plus courte possible.

Dans une architecture en couche et/ou distribué, il n'est pas envisageable de verrouiller les données lues et susceptibles d'être modifiées.

Par contre, il est souhaitable d'éviter de les mettre à jour de manière concurrente.

Dans Entity Framework, c'est la présence d'une colonne particulière gérant le numéro de version de la ligne dans la table qui sera mobilisé à cet effet.

Cette technique n'est pas l'exclusivité d'Entity Framework et nous l'avons déjà utilisée avec d'autres produits fonctionnant avec ADO.NET.

Cette colonne est de type RowVersion (DBTimestamp pour les anciennes versions de SQL Server). Il s'agit d'un nombre incrémentiel et non une valeur d'horodatage stockée sur 8 octets. La syntaxe DBTimestamp doit être évitée.

Si nous cherchons à modifier une ligne qui a été déjà modifiée par un autre processus entre le moment où nous avons extrait la ligne et celui où nous demandons sa mise à jour, nous obtiendrons une erreur de type DbConcurrency.

Par défaut, EF n'active pas le contrôle de modification concurrentielle. Il nous faut changer la propriété par le biais du concepteur de modèle d'entité (c'est aussi bien sur possible en code first).

Il s'agit de la propriété TS de l'entité Orders.



ComptoirAnglaisModel.Orders.TS Property	
<div> <div></div> <div></div> <div></div> </div>	
Facettes	
Longueur fixe	True
Longueur max.	8
Général	
Clé d'entité	False
Documentation	
Mode d'accès concurrentiel	Fixed

#### Etape 1 : Lecture et préparation de la modification

```
Orders commande = null;
using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Configuration.ProxyCreationEnabled = false;
    commande = dbContext.Orders.Find(11106);

    commande.RequiredDate = DateTime.Now.AddDays(10);
    commande.Etat = EntityState.Modified;
}
```

Autre processus : Mise à jour concurrente avec affichage des valeurs hexadécimales de la version de ligne.

```
SELECT OrderID, TS from [ComptoirAnglais].[dbo].[Orders]
where OrderID = 11106;
UPDATE
[ComptoirAnglais].[dbo].[Orders]
SET EmployeeID = 2
where OrderID = 11106;
SELECT OrderID, TS from [ComptoirAnglais].[dbo].[Orders]
where OrderID = 11106;
```

	OrderID	TS
1	11106	0x000000000000121D9

	OrderID	TS
1	11106	0x000000000000121DA

#### Etape 2 : Demande de Mise à jour

```
using (ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities())
{
    dbContext.Entry<Orders>(commande).State = (System.Data.Entity.EntityState)commande.Etat;
    dbContext.Database.Log = c => TraceurEF.Tracer("Modification Concurrentielle", c);
    dbContext.SaveChanges();
}
```

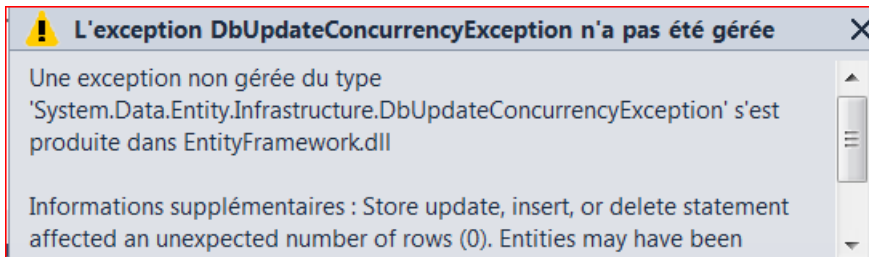
Regardons la trace de la demande :

Nous observons que l'activation de la mise à jour concurrentielle change la génération de la condition Where de l'instruction Update en ajoutant la valeur de la version de ligne

ORM Manipulation des objets

```
Connexion ouverte à 06/03/2016 17:20:39 +01:00
: Modification Concurrentielle
Début de la transaction à 06/03/2016 17:20:39 +01:00
: Modification Concurrentielle
UPDATE [dbo].[Orders]
SET [CustomerID] = @0, [EmployeeID] = @1, [OrderDate] = @2, [RequiredDate] = @3,
WHERE (([OrderID] = @4) AND ([TS] = @5))
SELECT [TS]
FROM [dbo].[Orders]
WHERE @@ROWCOUNT > 0 AND [OrderID] = @4: Modification Concurrentielle
```

Une exception est levée car l'instruction de mise à jour n'aboutit pas :



## 4.6 EXECUTION DIFFEREE OU IMMEDIATE

Nous pouvons envisager de préparer les requêtes sans les exécuter dans l'immédiat.

Nous approfondirons ce principe dans le chapitre consacré à Queryable et Enumerable.

Dans cet exemple, la requête d'interrogation est préparée en deux temps.

```
dbContext.Database.Log = c => TraceurEF.Tracer("Préparation Exécution", c);  
var requete = from c in dbContext.Customers  
              where (c.CompanyName.StartsWith("b"))  
              select (c);  
var requete2 = from c in requete  
               where (c.ContactName != null)  
               select (c);
```

Elle ne sera exécutée que lorsque nous demanderons à obtenir la liste des éléments répondant aux critères définis.

```
List<Customers> clients = requete2.ToList();
```

Cette exécution survient lors :

- d'une conversion vers une liste ou un tableau d'éléments (ToList, ToArray)
- lors d'une itération avec un cycle foreach qui provoque là encore l'élaboration d'une liste d'éléments
- l'extraction d'un élément first, last, single, ...

Dans la phase de préparation, les différentes expressions font évoluer la syntaxe de la requête.

Et la requête exécutée lors de la conversion en liste d'éléments prend bien en compte l'évolution de l'expression de la requête :

```
SELECT  
  [Extent1].[CustomerID] AS [CustomerID],  
  [Extent1].[CompanyName] AS [CompanyName],  
  [Extent1].[ContactName] AS [ContactName],  
  [Extent1].[ContactTitle] AS [ContactTitle],  
  [Extent1].[Address] AS [Address],  
  [Extent1].[City] AS [City],  
  [Extent1].[Region] AS [Region],  
  [Extent1].[PostalCode] AS [PostalCode],  
  [Extent1].[Country] AS [Country],  
  [Extent1].[Phone] AS [Phone],  
  [Extent1].[Fax] AS [Fax],  
  [Extent1].[IdPays2] AS [IdPays2]  
FROM [dbo].[Customers] AS [Extent1]  
WHERE ([Extent1].[CompanyName] LIKE 'b%') AND ([Extent1].[ContactName] IS NOT NULL)
```

Nous pourrions donc envisager de centraliser l'expression de nos requêtes au sein de classes externes pour les invoquer puis les faire évoluer en fonction d'un contexte d'exécution spécifique.

## 4.7 LA PROJECTION

Nous pouvons ne retenir dans notre sélection d'éléments que les propriétés qui nous intéressent.

Il faut même veiller à restreindre les attributs de nos requêtes aux seuls attributs utiles dans un souci d'efficacité et de préservation de bande passante sur le réseau.

La première requête retient l'ensemble des propriétés. Le résultat sera une liste d'entités de type Customers.

Les deux requêtes suivantes ne retiennent que certaines propriétés, dont les noms peuvent être redéfinis ou non. Les types sont anonymes.

La dernière requête s'appuie sur un type dérivé Client défini de manière statique.

```
ComptoirAnglaisEntities dbContext = new ComptoirAnglaisEntities();

List<Customers> clients = (from c in dbContext.Customers
    where (c.CompanyName.StartsWith("b"))
    select (c))
    .ToList<Customers>();

var anonyme1 = (from c in dbContext.Customers
    where (c.CompanyName.StartsWith("b"))
    select (new { c.CompanyName, c.ContactName, c.City }))
    .ToList();

var anonyme2 = (from c in dbContext.Customers
    where (c.CompanyName.StartsWith("b"))
    select (new { RaisonSociale = c.CompanyName, Contact = c.ContactName, Ville = c.City }))
    .ToList();

List<Client> derive = (from c in dbContext.Customers
    where (c.CompanyName.StartsWith("b"))
    select (new Client() { RaisonSociale = c.CompanyName, Ville = c.City }))
    .ToList<Client>();
```

Vous pouvez utiliser les méthodes de conversion génériques ou non.

**Les objets anonymes se consomment sur place.**

Requête 2 :

▷ [0]	{ CompanyName = "Boston", ContactName = "Frédéric Bost", City = "Sainte-Féréole" }
▷ [1]	{ CompanyName = "berglunds snabbköp", ContactName = "Christina Berglund", City = "Luleå" }
▷ [2]	{ CompanyName = "blauer see delikatessen", ContactName = "Hanna Moos", City = "Mannheim" }

Requête 3 :

▷ [0]	{ RaisonSociale = "Boston", Contact = "Frédéric Bost", Ville = "Sainte-Féréole" }
▷ [1]	{ RaisonSociale = "berglunds snabbköp", Contact = "Christina Berglund", Ville = "Luleå" }
▷ [2]	{ RaisonSociale = "blauer see delikatessen", Contact = "Hanna Moos", Ville = "Mannheim" }

Requête 4 :

▷ [0]	{LinqToEF_005.Client}
▷ [1]	{LinqToEF_005.Client}
▷ [2]	{LinqToEF_005.Client}

## 4.8 RECUPERER UN ELEMENT UNIQUE

Pour exprimer des requêtes simples comme le retour d'un élément unique, nous utiliserons les méthodes `single`, `SingleOrDefault`.

Autre possibilité est de récupérer un élément en position, premier, dernier.

```
Customers client = dbContext.Customers.  
    SingleOrDefault(c => c.CustomerID == "BOSTI");  
  
client = dbContext.Customers.ElementAtOrDefault(2);
```

### A noter :

Si nous utilisons la méthode **Single** et qu'aucun élément correspondant au filtre n'est renvoyé, nous obtenons une erreur (de même si plusieurs éléments sont renvoyés).

Si nous utilisons la méthode **SingleOrDefault** et qu'aucun élément ne fait partie du jeu de résultats, nous obtiendrons la valeur par défaut, **null** pour les types références.

Les autres méthodes qui permettent de renvoyer un élément unique fonctionnent de manière similaire. Voir le tableau des opérateurs de requête standards, catégorie **Elements**.

## 4.9 LES IMBRICATIONS

Nous pouvons utiliser des requêtes imbriquées pour éventuellement réaliser des filtres sur les entités connexes via les propriétés de navigation.

Nous utiliserons éventuellement la sélection avec un prédicat sur les collections ou sur un élément singulier.

Clients ayant des commandes en 2000

```
List<Customers> clients = (from c in dbContext.Customers  
    where c.Orders.Select(o => o.OrderDate.Value.Year == 2000 ).Count()>0  
    select c).ToList();
```

Commandes avec client dont nom de la compagnie commence par b

```
List<Orders> commandes = dbContext.Orders.  
    Where(o => o.Customers.CompanyName.StartsWith("b")).ToList();
```

Attention aux performances des requêtes corrélées, notamment si vous avez plusieurs niveaux d'imbrication. D'autres formes peuvent être plus efficaces.

## 4.10 LES AGREGATIONS

Les requêtes d'agrégation sont exécutées immédiatement pour produire les valeurs d'agrégats.

Il n'est pas possible de construire des agrégats sur des types anonymes.

Les agrégations sont souvent associées à des projections et des requêtes corrélées.

### 4.10.1 Agrégation sur entités imbriquées

Dans l'exemple qui suit, utilisation d'opérateurs d'agrégation, Sum et Count, sur des entités imbriquées. Cette approche est très usitée quand des données dérivées doivent être calculées.

Ici, connaissance du nombre de clients ayant commandé pour une somme supérieure à 25000

```
protected void ComptageClients()
{
    int nbClients = (from c in dbContext.Customers
                     where c.Orders.Sum(
                         o => o.Order_Details.Sum(
                             od => od.UnitPrice * od.Quantity)) >= 25000
                     select c).Count();

    txtResultat.Text
        += string.Format("Nombre de clients trouvés {0};", nbClients);
}
```

Imbrication: Customers → Orders → Order\_Details

Résultat : Nombre de clients trouvés 14;

## 4.11 LES REGROUPEMENTS

Comme en sql, l'instruction group by permet de définir un facteur de regroupement.

Dans l'exemple qui suit, nous récupérons le prix maximal par catégorie de produits.

Group By n'est pas exécuté immédiatement. Il existe un opérateur ToLookup qui permet d'exprimer des regroupements simples selon la syntaxe des méthodes uniquement.

```
var produitsplusChers = (from c in dbContext.Categories
                          join p in dbContext.Products on c.CategoryID equals p.CategoryID
                          group p by new { p.CategoryID, c.CategoryName } into cat

                          select new
                          {
                              cat.Key.CategoryID,
                              cat.Key.CategoryName,
                              cat,
                              nombreProduits = cat.Count(),
                              prixMax = cat.Max(p => p.UnitPrice)
                          });
```

Résultat :

```

Catégorie 1 beverages 2 Prix Max : 26,5200 nombre produits 12
Catégorie 2 sauces Prix Max : 30,0000 nombre produits 13
Catégorie 3 Desserts Prix Max : 1,0000 nombre produits 13
Catégorie 4 Dairy Products Prix Max : 1,0000 nombre produits 9
Catégorie 5 Grains/Cereals Prix Max : 1,0000 nombre produits 8
Catégorie 6 Meat/Poultry Prix Max : 1,0000 nombre produits 6
Catégorie 7 Produce Prix Max : 1,0000 nombre produits 5
Catégorie 8 Seafood Prix Max : 1,0000 nombre produits 12

```



Attention à la manière dont vous exprimez les facteurs de regroupement et la projection associée. Vous pouvez dégrader les performances de votre système si vous ne prêtez pas suffisamment attention à la forme de l'expression.

Soit la requête suivante qui doit établir le nombre de commandes par client :

```

dbContext.Database.Log = c => TraceurEF.Tracer("Groupes Version 1", c);
var req = from c in dbContext.Customers
          group c by new { c.CustomerID,
                          c.CompanyName,
                          nombreCdes = c.Orders.Count()
                          }
          into commandesParClient
          select commandesParClient;

```

Elle permet bien de récupérer le nombre de commandes par client comme le montre l'extrait résultat de l'exécution du cycle suivant :

```

foreach (var item in req)
{
    Console.WriteLine("Client : {0} {1} nombre de commandes : {2}",
        item.Key.CompanyName,
        item.Key.CustomerID,
        item.Key.nombreCdes
    );
}

```

```

Client : Boston A1962 nombre de commandes : 0
Client : TARTEMPION ALFKI nombre de commandes : 9
Client : ana trujillo emparedados y helados ANATR nombre de commandes : 4
Client : antonio moreno taquería ANTON nombre de commandes : 7
Client : around the horn AROUT nombre de commandes : 13

```

Toutefois la consultation du log nous montre une requête complexe et peu performante.

Après avoir réalisé le regroupement, d'autres sélections sont exécutées pour récupérer l'ensemble des attributs du client. Extrait *incomplet* pour exemple.

```

SELECT
    [Project3].[C1] AS [C1],
    [Project3].[CustomerID] AS [CustomerID],
    [Project3].[CompanyName] AS [CompanyName],
    [Project3].[C3] AS [C2],
    [Project3].[C2] AS [C3],
    [Project3].[CustomerID1] AS [CustomerID1],
    [Project3].[CompanyName1] AS [CompanyName1],
    [Project3].[ContactName] AS [ContactName],
    [Project3].[ContactTitle] AS [ContactTitle],
    [Project3].[Address] AS [Address],
    [Project3].[City] AS [City],
    [Project3].[Region] AS [Region],
    [Project3].[PostalCode] AS [PostalCode],
    [Project3].[Country] AS [Country],
    [Project3].[Phone] AS [Phone],
    [Project3].[Fax] AS [Fax],
    [Project3].[IdPays2] AS [IdPays2]
FROM ( SELECT
    [Project1].[CustomerID] AS [CustomerID],
    [Project1].[CompanyName] AS [CompanyName],
    1 AS [C1],
    [Project2].[CustomerID] AS [CustomerID1],
    [Project2].[CompanyName] AS [CompanyName1],
    [Project2].[ContactName] AS [ContactName],
    [Project2].[ContactTitle] AS [ContactTitle],
    [Project2].[Address] AS [Address],
    [Project2].[City] AS [City],
    [Project2].[Region] AS [Region],
    [Project2].[PostalCode] AS [PostalCode],
    [Project2].[Country] AS [Country],
    [Project2].[Phone] AS [Phone],
    [Project2].[Fax] AS [Fax],
    [Project2].[IdPays2] AS [IdPays2],
    CASE WHEN ([Project2].[CustomerID] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS [C2],
    [Project1].[C1] AS [C3]
FROM (SELECT
    [Extent1].[CustomerID] AS [CustomerID],
    [Extent1].[CompanyName] AS [CompanyName],
    (SELECT
        COUNT(1) AS [A1]
        FROM [dbo].[Orders] AS [Extent2]
        WHERE [Extent1].[CustomerID] = [Extent2].[CustomerID]) AS [C1]
    FROM [dbo].[Customers] AS [Extent1] ) AS [Project1]
LEFT OUTER JOIN (SELECT
    [Extent3].[CustomerID] AS [CustomerID],
    [Extent3].[CompanyName] AS [CompanyName],
    [Extent3].[ContactName] AS [ContactName],
    [Extent3].[ContactTitle] AS [ContactTitle],
    [Extent3].[Address] AS [Address],
    [Extent3].[City] AS [City],
    [Extent3].[Region] AS [Region],
    [Extent3].[PostalCode] AS [PostalCode],
    [Extent3].[Country] AS [Country],
    [Extent3].[Phone] AS [Phone],
    [Extent3].[Fax] AS [Fax],
    [Extent3].[IdPays2] AS [IdPays2]
FROM [dbo].[Customers] AS [Extent3] ) AS [Project2] ON [Project1].[CustomerID] = [Project2].[CustomerID]) AS [Project3]

```

Dans le jeu de résultat, il n'est pas mentionné de projection particulière.

Aussi, le service qui produit la requête considère que tous les champs issus de c (customers) doivent être présents dans le résultat.

Pour améliorer cette requête il nous suffit donc de réaliser une projection sur les seuls attributs présents dans le groupe.

Nous en profitons pour leur affecter une référence plus aisée.

```

var req = from c in dbContext.Customers
    group c by new
    {
        c.CustomerID,
        c.CompanyName,
        nombreCdes = c.Orders.Count()
    }
    into commandesParClient
    select new
    {
        CodeClient= commandesParClient.Key.CustomerID,
        NomClient = commandesParClient.Key.CompanyName,
        NombreCommandes= commandesParClient.Key.nombreCdes
    };

```



```
foreach (var item in req)
{
    Console.WriteLine("Client : {0} {1} nombre de commandes : {2}",
        item.CodeClient,
        item.NomClient,
        item.NombreCommandes
    );
}
```

Nous obtenons le même résultat attendu.

```
Client : A1962 Boston nombre de commandes : 0
Client : ALFKI TARTEMPION nombre de commandes : 9
Client : ANATR ana trujillo emparedados y helados nombre de commandes : 4
Client : ANTON antonio moreno taquería nombre de commandes : 7
Client : AROUT around the horn nombre de commandes : 13
```

Mais apprécions alors la simplicité de la requête !

```
SELECT
    1 AS [C1],
    [Project1].[CustomerID] AS [CustomerID],
    [Project1].[CompanyName] AS [CompanyName],
    [Project1].[C1] AS [C2]
FROM ( SELECT
    [Extent1].[CustomerID] AS [CustomerID],
    [Extent1].[CompanyName] AS [CompanyName],
    (SELECT
        COUNT(1) AS [A1]
        FROM [dbo].[Orders] AS [Extent2]
        WHERE [Extent1].[CustomerID] = [Extent2].[CustomerID]) AS [C1]
    FROM [dbo].[Customers] AS [Extent1]
) AS [Project1]: Groupes Version 1
```

## 4.12 LES JOINTURES

Il existe plusieurs manières de récupérer des éléments issus de plusieurs jeux d'entités.

Nous devons prêter attention à distinguer deux types de requêtes avec jointures :

- Les jointures groupées qui fournissent un résultat sous forme hiérarchique (arbre)
- Les jointures simples qui fournissent, comme le ferait SQL, un résultat sous forme de rectangle constitué de lignes et de colonnes.

### 4.12.1 Les jointures groupées

Nous souhaitons ici établir la liste des produits par catégorie.

Nous allons définir un groupe de produits pour chaque catégorie. Ce type de jointure produira des ensembles hiérarchiques de données.

Nous pouvons constituer ce groupe de jointure en utilisant le mot clé **into** associé à **join** en utilisant l'approche par requêtes.

L'utilisation du mot clé **into** provoque la création d'un groupe ici désigné sous le terme prodParCat

```
var gCat_Prod = from c in dbContext.Categories
                join p in dbContext.Products on c.CategoryID equals p.CategoryID
                into prodParCat
                select new { cat = c, produits = prodParCat }
                ;|
```

Les données se présenteront ainsi :

Cat	ProdParCat		
Categorie 1	Produit 1	Produit 2	Produit3
Categorie 2	Produit 4	Produit 5	
Categorie 3			

Pour présenter les données, nous réaliserons deux boucles.

La boucle Externe qui permettra d'extraire les catégories

La boucle Interne qui permettra d'extraire chaque produit présent dans la catégorie

```
foreach (var categorie in gCat_Prod)
{
    Console.WriteLine("Catégorie {0} {1}", categorie.cat.CategoryID, categorie.cat.CategoryName);
    foreach (var produit in categorie.prodParCat)
    {
        Console.WriteLine("Produit {0} {1}", produit.ProductID, produit.ProductName);
    }
}
```

Vous pouvez aussi exprimer le groupe de jointure en utilisant la méthode **GroupJoin** et les expressions lambda.

```
var gCat_Prod2 = dbContext.Categories
    .GroupJoin(dbContext.Products,
        c => c.CategoryID,
        p => p.CategoryID,
        (categories, produits)
        => new { categories, produits });
```

Première Séquence  
Deuxième séquence à joindre  
Attribut de jointure Seq 1  
Attribut de jointure Seq  
Collection 2<sup>ème</sup> séquence  
Elément 1<sup>ère</sup> séquence

Cette forme donnera exactement le même résultat que la précédente.

#### 4.12.2 Les jointures simples

Reprenons l'exemple précédent et constituons une jointure simple.

```
ComptoirAnglaisDataContext dcComptoirAnglais = new ComptoirAnglaisDataContext();
dcComptoirAnglais.Log = Console.Out;
var prod_cat = from c in dcComptoirAnglais.Categories
    join p in dcComptoirAnglais.Products on c.CategoryID equals p.CategoryID
    select new {categories= c, produits = p };
```

Les données se présenteront ainsi :

Categories	produits
Categorie 1	Produit 1
Categorie 1	Produit 2
Categorie 1	Produit 3
Categorie 2	Produit 4

Pour afficher les données, une seule boucle :

ORM Manipulation des objets

```
foreach (var cat_prod in prod_cat)
{
    Console.WriteLine("Catégorie {0} {1}", cat_prod.categories.CategoryID,
        cat_prod.categories.CategoryName);
    Console.WriteLine("Produit {0} {1}", cat_prod.produits.ProductID,
        cat_prod.produits.ProductName);
}
```

Mais nous avons bien entendu perdu la notion de regroupement !

#### 4.12.3 Remarques sur les jointures

Pour obtenir ici les personnes sans adresse, ce qui serait équivalent à une jointure gauche, nous pouvons recourir à la syntaxe des requêtes :

```
ContexteAFPADDataContext dc = new ContexteAFPADDataContext();
var personnes = from pers in dc.Personne
                join
                adresses in dc.Personne_Adresse
                on pers equals adresses.Personne into psa
                from adressesPersonnes in psa.DefaultIfEmpty()
                where pers.Nom.StartsWith("B")
                select new { pers.Nom, pers.Prenom, adressesPersonnes.AdressePostale.CodePostal };
List<object> res = personnes.ToList<object>();
```

Dans Adresses personnes se retrouveront les personnes ayant ou non une adresse.

Il est aussi possible d'utiliser la syntaxe des méthodes comme l'illustre l'exemple suivant.

Dans la requête suivante, j'utilise la méthode Group Join qui réalise de base une jointure gauche depuis l'entité sur laquelle elle porte vers l'entité désignée dans le regroupement.

Cette requête réalise un comptage des adresses de chaque personne.

```
AfpaDataContext dc = new AfpaDataContext();

var req = dc.Personne.GroupJoin(dc.Personne_Adresse,
    p => p.IDPersonne,
    ap => ap.IdPersonne,
    (p, pa) => new { p.Prenom, p.Nom, NbAd = pa.Count() });
foreach (var item in req)
{
    Console.WriteLine("La personne {0} {1} a {2} adresses ", item.Prenom, item.Nom, item.NbAd);
}
```

Linq permet l'interrogation de données hiérarchiques.

Par défaut, les enfants d'une entité parente ne sont chargés qu'au dernier moment.

Mais nous pouvons modifier ce comportement afin de charger les enfants dans le même temps en modifiant les DataLoadOptions du modèle de contexte.

```
AfpaDataContext dc = new AfpaDataContext();
DataLoadOptions Options = new DataLoadOptions();
Options.LoadWith<Personne>(p => p.Personne_Adresse);
Options.LoadWith<Personne_Adresse>(p => p.AdressePostale);
dc.LoadOptions = Options;
var PersonnesAdresses = (from p in dc.Personne
    where p.Nom.StartsWith(txtNom.Text)
    select p)
    .Take(5)
    .ToList();
foreach (var item in PersonnesAdresses)
{
    AdressePostale ad = (item.Personne_Adresse.FirstOrDefault() == null) ?
        null :
        item.Personne_Adresse.First().AdressePostale;
}
```

### 4.13 LES PARTITIONS

Les opérateurs de partitionnement nous seront fort utiles, comme par exemple pour mettre en œuvre des mécanismes de pagination de données.

Certains d'entre vous ont peut-être déjà mis en place de tels mécanismes en SQL avec ou sans recours aux expressions de table commune.

Nous allons ici apprécier la simplicité de l'expression Linq.

Nous aurons ici besoin de deux variables de type intégral pour retourner les objets de la page courante.

- numeroPageCourante.
- taillePage : nombre d'objets par page

Et de deux opérateurs :

- Skip(x) : permet de passer outre les x premiers éléments de la liste
- Take(y) : permet de retenir y éléments max dans la liste

La requête linq :

```
int pageCourante = 2;
int taillePage = 5;
List<Customers> pageClients = dbContext.Customers
    .Where(c => c.CompanyName.StartsWith("b"))
    .OrderBy(c => c.CompanyName)
    .Skip(pageCourante * taillePage - taillePage)
    .Take(taillePage)
    .ToList();
```

Requête SQL générée :

```
SELECT TOP (5)
    [Filter1].[CustomerID] AS [CustomerID],
    [Filter1].[CompanyName] AS [CompanyName],
    [Filter1].[ContactName] AS [ContactName],
    [Filter1].[ContactTitle] AS [ContactTitle],
    [Filter1].[Address] AS [Address],
    [Filter1].[City] AS [City],
    [Filter1].[Region] AS [Region],
    [Filter1].[PostalCode] AS [PostalCode],
    [Filter1].[Country] AS [Country],
    [Filter1].[Phone] AS [Phone],
    [Filter1].[Fax] AS [Fax],
    [Filter1].[IdPays2] AS [IdPays2]
FROM ( SELECT [Extent1].[CustomerID] AS [CustomerID], [Extent1].[CompanyName] AS
    FROM [dbo].[Customers] AS [Extent1]
    WHERE [Extent1].[CompanyName] LIKE 'b%'
) AS [Filter1]
WHERE [Filter1].[row_number] > 5
ORDER BY [Filter1].[CompanyName] ASC: Pagination
```



Attention à l'ordre dans lequel vous invoquez les opérateurs de partitionnement.

## 5. CHARGEMENT IMMEDIAT OU DIFFERE

Il est important de bien comprendre ces mécanismes pour conserver de bonnes performances à votre application et maîtriser le code SQL généré.

### 5.1 LE CHARGEMENT DIFFERE (LAZYLOADING)

Le chargement est retardé jusqu'à ce qu'un accès à l'information soit demandé.

Le problème est que nous ne maîtrisons pas nécessairement quand il aura lieu.

J'ai par le passé été confronté à des problèmes cocasses comme, avec une application WPF, le chargement des données l'une liste lorsque le contrôle lié à ces dernières se dessinait !

Du plus bel effet pour l'utilisateur qui se demande ce qui se passe car, dans ce contexte, il y aura autant de requêtes que d'occurrences de contrôles dans la liste et de rafraîchissement de la surface du contrôle.

**Avec la dernière version d'Entity Framework Core, ce modèle de chargement n'est plus disponible par défaut et c'est plutôt une bonne chose.**

Je peux simuler un tel exemple en exécutant le code suivant qui affiche la date de dernière commande d'un client ou l'expression aucune s'il n'a passé aucune commande.

```
var clients = dbContext.Customers.Where(c => c.CompanyName.StartsWith("F"));

foreach (Customers item in clients)
{
    Console.WriteLine("Client : {0} {1} Date dernière commande {2}",
        item.CustomerID, item.CompanyName,
        (item.Orders.Count() == 0 ? "Aucune" : item.Orders.FirstOrDefault().OrderDate.ToString()));
}
```

```
SELECT
    [Extent1].[CustomerID] AS [CustomerID],
    [Extent1].[CompanyName] AS [CompanyName],
    [Extent1].[ContactName] AS [ContactName],
    [Extent1].[ContactTitle] AS [ContactTitle],
    [Extent1].[Address] AS [Address],
    [Extent1].[City] AS [City],
    [Extent1].[Region] AS [Region],
    [Extent1].[PostalCode] AS [PostalCode],
    [Extent1].[Country] AS [Country],
    [Extent1].[Phone] AS [Phone],
    [Extent1].[Fax] AS [Fax],
    [Extent1].[IdPays2] AS [IdPays2]
FROM [dbo].[Customers] AS [Extent1]
WHERE [Extent1].[CompanyName] LIKE 'F%': Chargement différé
```

Autant de requêtes que d'objets customers.

```

SELECT
    [Extent1].[OrderID] AS [OrderID],
    [Extent1].[CustomerID] AS [CustomerID],
    [Extent1].[EmployeeID] AS [EmployeeID],
    [Extent1].[OrderDate] AS [OrderDate],
    [Extent1].[RequiredDate] AS [RequiredDate],
    [Extent1].[ShippedDate] AS [ShippedDate],
    [Extent1].[ShipVia] AS [ShipVia],
    [Extent1].[Freight] AS [Freight],
    [Extent1].[ShipName] AS [ShipName],
    [Extent1].[ShipAddress] AS [ShipAddress],
    [Extent1].[ShipCity] AS [ShipCity],
    [Extent1].[ShipRegion] AS [ShipRegion],
    [Extent1].[ShipPostalCode] AS [ShipPostalCode],
    [Extent1].[ShipCountry] AS [ShipCountry],
    [Extent1].[TS] AS [TS]
FROM [dbo].[Orders] AS [Extent1]
WHERE [Extent1].[CustomerID] = @EntityKeyValue1: Chargement différé

```

Le résultat est correct mais il nécessite beaucoup trop d'échanges avec le serveur.

```

Client : FAMIA familia arquibaldo Date dernière commande 31/10/2007 00:00:00
Client : FISSA fissa fabrica inter. salchichas s.a. Date dernière commande Aucune
Client : FOLIG folies gourmandes Date dernière commande 22/12/2007 00:00:00
Client : FOLKO folk och få hb Date dernière commande 27/04/2008 00:00:00
Client : FRANK frankenversand Date dernière commande 09/04/2008 00:00:00
Client : FRANR france restauration Date dernière commande 24/03/2008 00:00:00
Client : FRANS franchi s.p.a. Date dernière commande 30/04/2008 00:00:00
Client : FURIB furia bacalhau e frutos do mar Date dernière commande 19/03/2008 00:00:00

```

💣 Pour qu'il soit envisageable d'utiliser le chargement différé la configuration du contexte de données doit autoriser :

- Le chargement différé LazyLoading = true
- Le recours aux proxys proxyEnabled = true
- Des propriétés de navigation en accès public et autorisant la substitution virtual (par des types dérivés proxys)

Le chargement différé doit être évité car il induit de nombreux problèmes de performances et des effets de bord.

## 5.2 LE CHARGEMENT IMMEDIAT (EAGERLOADING)

Nous allons préciser, au moment de l'expression de la requête, quelles entités connexes charger.

Le chargement immédiat permet de totalement maîtriser le processus d'extraction des entités associées et est ainsi de loin préférable au chargement différé. Il incombe par contre au développeur d'être vigilant dans l'expression des demandes d'interrogation de la base de données.

Ainsi, dans le cadre d'un dialogue classique maître esclave où l'on sélectionne un élément dans une liste pour afficher le détail des éléments connexes (ex : client-commandes), nous chargerons les données des commandes après le choix d'un client.

Il n'existe pas de règle absolue.

L'expression des requêtes sera déterminée en fonction des volumes de données et de la fréquence des demandes.

La méthode **Include** nous permet de préciser les éléments connexes devant être chargés.

```
client = dbContext1.Customers
.Include(c => c.Orders)
.FirstOrDefault(c => c.CustomerId == "ALFKI");
```

Il peut y avoir plusieurs méthodes **Include** sollicitées.

La méthode **ThenInclude** permet d'inclure les éléments de second niveau. Ici, le produit retenu dans le détail d'une ligne de commande.

```
var commandes = dbContext1.Orders
    .Include(c => c.OrderDetails).ThenInclude(o => o.Product)
    .Include(c => c.Employee)
    .Include(c => c.Customer)
    .FirstOrDefault(c => c.CustomerId == "ALFKI");
```

Il est aussi possible de recourir à un chargement via la méthode **Entry**

**Pour une propriété de navigation de type Collection (pluriel)**

```
dbContext1.Entry(client)
    .Collection(o => o.Orders)
    .Load();
```

**Pour une propriété de navigation de type Reference (singulier)**

```
dbContext1.Entry(commandes)
    .Reference(o => o.Customer);
```



## 6. ARBORESCENCE D'EXPRESSION IQUERYABLE, IENUMERABLE

### 6.1 ARBORESCENCE D'EXPRESSION LAMBDA

Les arborescences d'expression lambda diffèrent des expressions lambda dans la mesure où elles ne pointent pas vers une fonction mais sont une représentation de l'action qui sera exécutée.

Les arborescences d'expression sont de type `Linq.Expressions.Expression`.

Elles doivent être **compilées** pour être exécutées dans le **langage ad'hoc**.

Exemple avec délégué défini et délégué générique :

```
delegate int operationArithmetique(int i);  
1 référence  
private void btnArborescenceExpLambdas_Click(object sender, EventArgs e)  
{  
    System.Linq.Expressions.Expression<operationArithmetique> operation = x => x * x;  
    System.Linq.Expressions.Expression<Func<int, int>> operation2 = x => x * x;  
    int resultat = operation.Compile()(3);  
    resultat = operation2.Compile()(3);  
}
```

### 6.2 IQUERYABLE ET IENUMERABLE

Nous retrouvons ces distinctions au niveau de Linq et Linq to Entities et allons démontrer les incidences au niveau de nos développements.

Observons deux requêtes très semblables à première vue :

```
var clientsQ = (from c in dbContext.Customers  
                select c)  
                .Where(c => c.CompanyName.StartsWith("b"))  
                .ToList();
```

**Where** prend un paramètre de type **System.Linq.Expressions.Expression** et viendra modifier l'arbre d'expressions Linq et impactera la requête SQL produite qui agit sur un type qui implémente l'interface **IQueryable**. Cet arbre d'expression sera ensuite compilé pour produire le code SQL.

(extension) IQueryable<Customers> IQueryable<Customers>.Where<Customers>(System.Linq.Expressions.Expression<Func<Customers,bool>> predicate)  
Filtre une séquence de valeurs selon un prédicat.

```
var clientsE = (from c in dbContext.Customers  
                select c)  
                .ToList()  
                .Where(c => c.CompanyName.StartsWith("b"));
```

**Where** la **méthode** dont le corps a été défini dans l'expression lambda sera exécutée sur l'énumération.

(extension) IEnumerable<Customers> IEnumerable<Customers>.Where<Customers>(Func<Customers,bool> predicate)  
Filtre une séquence de valeurs selon un prédicat.

Ces deux formes de requêtes vont au final produire le même résultat ! Seulement pas dans les mêmes conditions. Observons le code SQL généré

```
SELECT
  [Extent1].[CustomerID] AS [CustomerID],
  [Extent1].[CompanyName] AS [CompanyName],
  [Extent1].[ContactName] AS [ContactName],
  [Extent1].[ContactTitle] AS [ContactTitle],
  [Extent1].[Address] AS [Address],
  [Extent1].[City] AS [City],
  [Extent1].[Region] AS [Region],
  [Extent1].[PostalCode] AS [PostalCode],
  [Extent1].[Country] AS [Country],
  [Extent1].[Phone] AS [Phone],
  [Extent1].[Fax] AS [Fax],
  [Extent1].[IdPays2] AS [IdPays2]
FROM [dbo].[Customers] AS [Extent1]
WHERE [Extent1].[CompanyName] LIKE 'b%': IQ
```

```
SELECT
  [Extent1].[CustomerID] AS [CustomerID],
  [Extent1].[CompanyName] AS [CompanyName],
  [Extent1].[ContactName] AS [ContactName],
  [Extent1].[ContactTitle] AS [ContactTitle],
  [Extent1].[Address] AS [Address],
  [Extent1].[City] AS [City],
  [Extent1].[Region] AS [Region],
  [Extent1].[PostalCode] AS [PostalCode],
  [Extent1].[Country] AS [Country],
  [Extent1].[Phone] AS [Phone],
  [Extent1].[Fax] AS [Fax],
  [Extent1].[IdPays2] AS [IdPays2]
FROM [dbo].[Customers] AS [Extent1]: IQuery
```

Dans le premier cas de figure, la requête a été générée à partir de l'arbre d'expression modifié par l'expression linq et le code sql prend donc en compte le filtre.

Dans le deuxième cas, le filtre s'applique sur le résultat de la requête SQL exécutée qui ne comporte aucun filtre. Le filtre sera appliqué sur les objets en mémoire.

Les performances ne seront absolument pas les mêmes ! Notamment dans le cadre d'architectures distribuées.

Dans le deuxième cas, nous allons acheminer vers l'application cliente de nombreux clients dont nous n'avons que faire...

Nous allons ensuite itérer sur l'ensemble des éléments de cette séquence pour ne retenir que ceux qui satisfassent au filtre.

Ci-dessous le fonctionnement du type implémentant IEnumerable.

Le type IEnumerable permet d'itérer sur un ensemble d'éléments et fonctionne avec un énumérateur qui permet de se déplacer dans une séquence et adresser l'élément courant.

Il s'agit d'un mécanisme similaire à celui mis en œuvre dans le cadre d'un curseur de base de données.

Il est implémenté de base dans tous les éléments de type List, Collection, Array.

Il s'alimente via un bloc itératif de type **foreach** au sein duquel l'élément sélectionné de la liste est retourné. Le traitement de cette séquence se fait grâce au mot clé **yield return**.

L'exemple d'extension d'IEnumerable ci-dessous reproduit ce mécanisme :

```
public static partial class IEnumerable
{
    public static IEnumerable<T> test<T>(this IEnumerable<T> sequence, Func<T, bool> condition)
    {
        foreach (T item in sequence)
        {
            if (condition(item)) yield return item;
        }
    }
}
```

```
static void Main(string[] args)
{
    int[] entiers = new int[]{1,3,4,10,11,65};
    IEnumerable<int> listePairs = IEnumerable.test<int>(entiers, e => e % 2 == 0);
    listePairs.ToList();
}
```

Si l'objet fourni dans la liste d'**entiers** est pair, il sera alors retenu dans la collection produite en sortie par **test**.

### 6.3 CONCLUSION

Attention donc à veiller à bien différencier les deux natures d'expressions et vérifier le type sur lequel nous travaillons. Soyons particulièrement vigilant si nous devons construire nos requêtes en plusieurs étapes.

## 7. EXECUTER DES REQUETES SQL BRUTES

### 7.1 EXECUTION DE REQUETES DE SELECTION AVEC OU SANS PARAMETRES

Il est toujours possible de recourir à du code SQL pour produire nos entités et ne pas recourir aux mécanismes de transformation de base.

```
var clients = dbContext.Customers.FromSqlRaw(
    "Select * from Customers")
    .ToList();
```

### Toujours utiliser le paramétrage pour les requêtes SQL brutes

Lorsque vous introduisez des valeurs fournies par l'utilisateur dans une requête SQL brute, vous devez veiller à éviter les attaques par injection SQL. En plus de valider le fait que ces valeurs ne contiennent pas de caractères non valides, utilisez toujours le paramétrage qui envoie les valeurs séparées du texte SQL.

En particulier, ne transmettez jamais une chaîne concaténée ou interpolée (\$) avec des valeurs non validées fournies par l'utilisateur dans **FromSqlRaw** ou **ExecuteSqlRaw**.

Les méthodes **FromSqlInterpolated** et **ExecuteSqlInterpolated** permettent d'utiliser la syntaxe d'interpolation de chaîne d'une manière qui protège contre les attaques par injection SQL.

Exemple :

Ici, `companyName` sera automatiquement transformé en `DbParameter` évitant ainsi toute injection SQL.

```
clients = dbContext.Customers.
    FromSqlInterpolated($"Select * from Customers where companyname like {companyName}")
    .ToList();
```

D'autres mécanismes vous permettent d'utiliser les procédures stockées.

## 8. OPERATEURS DE REQUETE STANDARD

### Agrégation

Aggregate	Exécute une méthode personnalisée sur une séquence
Average	Calcule la moyenne d'une séquence de valeurs numériques
Count	Renvoie le nombre d'éléments d'une séquence sous la forme d'un entier
LongCount	Renvoie le nombre d'éléments d'une séquence sous la forme d'un nom long
Min	Recherche le plus petit nombre d'une séquence de nombres
Max	Recherche le plus grand nombre d'une séquence de nombres
Sum	Additionne les nombres d'une séquence

### Concaténation

Concat	Concatène deux séquences en une seule
--------	---------------------------------------

### Conversion

Cast	Convertit les éléments d'une séquence dans un type donné
OfType	Filtre les éléments d'une séquence d'un type donné
ToArray	Renvoie un tableau à partir d'une séquence
ToDictionary	Renvoie un dictionnaire à partir d'une séquence
ToList	Renvoie une liste à partir d'une séquence
ToLookup	Renvoie une recherche à partir d'une séquence
ToSequence	Renvoie une séquence IEnumerable

### Élément

DefaultIfEmpty	Crée un élément par défaut pour une séquence vide
ElementAt	Renvoie l'élément à un index donné dans une séquence
ElementAtOrDefault	Renvoie l'élément à un index donné dans une séquence une valeur par défaut si l'index est en dehors de la plage
First	Renvoie le premier élément d'une séquence
FirstOrDefault	Renvoie le premier élément d'une séquence ou une valeur par défaut si aucun élément n'est trouvé
Last	Renvoie le dernier élément d'une séquence

LastOrDefault	Renvoie le dernier élément d'une séquence ou une valeur par défaut si aucun élément n'est trouvé
Single	Renvoie l'élément unique d'une séquence
SingleOrDefault	Renvoie l'élément unique d'une séquence ou une valeur par défaut si aucun élément n'est trouvé

## Égalité

SequenceEqual	Compare deux séquences pour voir si elles sont équivalentes
---------------	---

## Génération

Empty	Génère une séquence vide
Range	Génère une séquence pour une plage donnée
Repeat	Génère une séquence en répétant un élément x fois

## Regroupement

GroupBy	Regroupe les éléments d'une séquence en fonction d'un regroupement donné
---------	--

## Jointure

GroupJoin	Exécute une jointure regroupée sur deux séquences
Join	Exécute une jointure intérieure sur deux séquences

## Tri

OrderBy	Trie une séquence par valeur(s) dans l'ordre croissant
OrderByDescending	Trie une séquence par valeur(s) dans l'ordre décroissant
ThenBy	Trie une séquence déjà triée dans l'ordre croissant
ThenByDescending	Trie une séquence déjà triée dans l'ordre décroissant
Reverse	Inverse l'ordre des éléments d'une séquence

## Partitionnement

Skip	Renvoie une séquence qui ignore un nombre donné d'éléments
SkipWhile	Renvoie une séquence qui ignore des éléments ne répondant pas à une expression
Take	Renvoie une séquence qui prend un nombre donné d'éléments

TakeWhile	Renvoie une séquence qui prend des éléments répondant à une expression
-----------	--

## Projection

Select	Crée une projection de parties d'une séquence
SelectMany	Crée une projection « un à plusieurs » de parties d'une séquence

## Quantificateurs

All	Détermine si tous les éléments d'une séquence vérifient une condition
Any	Détermine si des éléments d'une séquence remplissent une condition
Contains	Détermine si une séquence contient un élément donné

## Restriction

Where	Filtre les éléments d'une séquence
-------	------------------------------------

## Série

Distinct	Renvoie une séquence sans éléments dupliqués
Except	Renvoie une séquence différence entre deux séquences
Intersect	Renvoie une séquence représentant l'intersection de deux séquences
Union	Renvoie une séquence représentant l'union de deux séquences

## Inclusion

Include	Inclut des éléments connexes (enfants) dans la requête
ThenInclude	Inclut des éléments de 2 <sup>ème</sup> niveau Include(n1).ThenInclude(n2)

## **CREDITS**

### **ŒUVRE COLLECTIVE DE L'AFPA**

**Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services**

### **Equipe de conception (IF, formateur, mediatiseur)**

Vincent Bost – Formateur Etudes et Développement Informatique

**Date de mise à jour : 12/01/2020**

## **Reproduction interdite**

Article L. 122-4 du code de la propriété intellectuelle.

« Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droits ou ayants cause est illicite. Il en est de même pour la traduction, l'adaptation ou la reproduction par un art ou un procédé quelconque. »

ORM Manipulation des objets

Afpa © 2020 – Section Tertiaire Informatique – Filière « Etude et développement »