Tome 3

Linq, Entity Framework & RIA Services

Olivier Dahan

Microsoft Most Valuable Professional

Collection
ALL DOT BLOG

e-naxos

www.e-naxos.com

Formation – Audit – Conseil – Développement
XAML (Windows Store, WPF, Silverlight, Windows Phone), C#
Cross-plateforme Windows / Android / iOS
UX Design

ALL DOT.BLOG

Tome 3

Linq, Entity Framework & RIA Services

Tout Dot.Blog par thème sous la forme de livres PDF gratuits !

Olivier Dahan
odahan@gmail.com

Table des matières

LINQ 5
Ce qui fait de LINQ une nouveauté 5
LINQ to Objects 7
LINQ to ADO.NET 11
LINQ to XML 12
Les exemples 12
Conclusion 53
Class Helper, LINQ et fichiers CSV 53
Le problème à résoudre 53
La méthode la plus simple 53
Une méthode plus complète 55
Conclusion 57
LINQ to {tapez ce que vous voulez ici}. Des providers à foison ! 58
Conclusion 59
Mettre des données en forme en une requête LINQ 60
Traiter un flux RSS en 2 lignes ou "les trésors cachés de .NET 3.5" 61
LINQ et Réflexion pour obtenir les valeurs de System.Drawing.Color 63
Un cadeau à télécharger : des exemples LINQ ! 64
Linq au quotidien, petites astuces et utilisations détournées 66
Initialiser un tableau, une liste 67
Plusieurs itérations en une seule 67
Créer des séquences aléatoires 68
Créer des chaînes de caractères 68
Convertir une valeur en séquence de 1 objet 69
Enumérer tous les sous-ensembles d'un tableau 69
Un peu de code 70
Conclusion 70
Donner du peps à Linq ! (Linq dynamique et parser d'expressions) 70

LINQ à toutes les sauces ! 73
Cas 1 : Lister les services actifs de Windows. 73
Cas 2 : Remettre à unchecked tous les Checkbox d'une form 73
LINQ Join sur plusieurs champs 74
Join on / Equals 74
Mais avec deux champs ? 75
L'astuce 76
Linq to Entities : ".Date" non supporté – Une solution 77
Séparons bien les problèmes 77
Un avertissement pour commencer 78
Revenons au problème 78
La solution 79
Chat échaudé 80
Conclusion 80
Créer un arbre des répertoires avec XML (et l'interroger avec LINQ to XML) 80
ADO.Net Entity Framework - une série d'articles en US 84
Entity Framework Application Patterns. Résumé de la conférence DAT303 Teched 86
Donc de quoi s'agissait-il ? 86
Change tracking et Identity resolution 87
Des entités qui passent les frontières 87
Compilation LINQ 88
Conclusion 88
Entity Framework - Résumé de la session DAT201 aux TechEd 88
L'Entity Framework 89
L'accès aux données de ADO.NET 2.0 89
L'accès aux données avec ADO.NET Entity Framework 91
L'évolution dans les accès aux données 92
L'Entity Data Model (EDM) 93
Interroger les données 95
LINQ to Entities 95

Entity SQL 96
La traduction des requêtes 97
EntityClient, le fournisseur d'accès aux données de EF 97
Les métadonnées 98
Conclusion 98
Entity Framework et la compatibilité arrière (problème Datetime2). 99
La solution 99
Conclusion 100
Entity Framework : Include avec des jointures, inclure réellement les données 100
Eager Loading 100
Quel est le problème ? 101
Solutions 102
Conclusion 103
Modifier le Timeout des Wcf Ria Services 103
Timeout 103
Un petit plus long s'il vous plait M. Cadbury ! 104
Une classe outil pour modifier le Timeout 105
Deux options d'utilisation 106
Conclusion 108
WCF Ria Services : Charger les entités associées 108
Des entités et des associations 108
Interrogation des données avec les entités associées 109
Autoriser le chargement des entités associées 110
Conclusion 111
Ria Services, MVVM Light, Silverlight et données de conception (design time data) – Astuces 111
Avertissements 113
E-Naxos 113

LINQ

Language-Integrated Query, ou Requêtage integré au langage, est un des ajouts les plus marquants du Framework 3.5 et de C# 3.0. Et les mots sont pesés.

Très raisonnablement l'avancée conceptuelle derrière LINQ est un bouleversement énorme au moins aussi important que le Framework .NET lui-même à sa sortie avec toutes ses technologies et ses outils.

> *Vous aidez à prendre la mesure de ce bouleversement, c'est le but du présent article écrit **lors de la sortie de LINQ**, il y a donc quelques années. Ce décalage dans le temps ne gâche rien à cette introduction puisque LINQ est resté le même, seule LINQ to Entities a beaucoup évolué depuis.*

Une précision : Ce qui va suivre n'est pas un cours sur LINQ, plus humblement c'est un vaste tour d'horizon de cette technologie, à la fois par des explications sur le « pourquoi » et le « comment » et par des exemples de code. Je reviendrai dans d'autres articles plus en détail sur certains aspects. Pour l'instant, venez découvrir pourquoi il y aura un avant LINQ et un après LINQ et pourquoi vous refuserez de développer dans le futur avec des langages ne le supporteront pas…

Ce qui fait de LINQ une nouveauté

Si vous avez une vague idée de ce qu'est LINQ vous pensez peut-être que LINQ n'est finalement qu'un nouveau procédé d'O/R mapping (ORM) comme il en existe de longue date sous .NET ou sous Java, et même sous Delphi avec ECO.

Cette approche est fausse à plus d'un titre !

Tout d'abord LINQ n'est pas un utilitaire, un framework lourd à maîtriser, c'est une évolution naturelle de la syntaxe de C#, il fait donc partie intégrante du langage (et de la plateforme .NET qui fournit les services). Ce n'est ni un ajout, ni une verrue, ni une astuce, c'est C# 3.0, tout simplement.

Ensuite, LINQ n'est en rien un simple outil d'ORM, c'est plutôt un SQL totalement objet qui s'adapte à différents contextes, dont les bases de données mais pas seulement. Car même cette comparaison est réductrice, LINQ est bien plus simple et plus puissant par exemple que l'OCL utilisé par ECO de Borland ou que les différentes versions de Object-SQL implémentées dans certains SGBD-Orienté Objet comme OQL dans O2 par exemple. Même NHibernate possède aujourd'hui un LINQ to

NHibernate malgré le succès de l'outil et de sa philosophie. Quant à DB4O, elle propose quelque chose de plus utilisable, d'où son relatif succès d'estime mais tout comme la base MySQL la pub parle de gratuité, mais en réalité les licences « pros » sont onéreuses… Gardez vos sous ! Tout cela coûte plus cher qu'un indispensable abonnement à MSDN qui vous offre Visual Studio (et bien plus) avec son LINQ intégré !

LINQ est donc très différent de tout cela. Pourquoi ? Parce que tous ces langages plus ou moins intégrés à des EDI, des frameworks ou à certains SGBD-O sont totalement disjoints du langage de programmation principal utilisé par le développeur, ils ont une syntaxe propre et non cohérente avec celle du langage des applications complexifiant la conception de ces dernières et forçant une gymnastique intellectuelle à la charge du développeur pour faire le lien entre les deux mondes… On échange donc un mapping O/R pour un mapping encore plus complexe entre deux systèmes objets incompatibles ! LINQ évite tous ces problèmes, LINQ c'est du C# (ou du VB), ni plus ni moins.

LINQ est une aussi une continuité logique vers plus d'objectivation des données, mouvement entamé depuis longtemps sous .NET. On se rappellera par exemple de l'introduction dans le Framework 2.0 de l'`ObjectDatasource` permettant d'utiliser une grappe d'objets en mémoire comme source de données pour des objets d'interface sous ASP.NET, ou même du DataBinding de .NET 1.0 qui fait que tout objet avec ses propriétés peut être vu comme une source de données. Ces progrès étaient immenses (comparez le DataBinding de .NET 1.0 avec la complexité de créer un composant lié à une source de données sous VC++ et la MFC, Delphi ou VB Win32 par exemple…) mais on pouvait aller plus loin.

LINQ est, de fait, l'aboutissement de ce long voyage vers plus d'abstraction et en même temps vers plus de pragmatisme. LINQ est tout sauf compliqué, il est simple et naturel. Mais ce qu'il permet de faire est d'une incroyable puissance.

Pourquoi LINQ ? Simplement parce qu'un logiciel passe son temps à manipuler des données (au sens large) et qu'il était logique qu'un jour les langages informatiques intègrent enfin ces dernières à leur fonctionnement. Faire de sources XML ou SQL, et même de listes d'objets, des « citoyens à part entière » du langage, c'est ça LINQ : arrêter la schizophrénie entre données du langage (entiers, chaînes, collections…) et données tout court (le plus souvent persistantes sur un SGBD mais pas seulement) en unifiant la syntaxe de manipulation quelle que soit la source.

LINQ est ainsi utilisable dans plusieurs contextes dont le développeur n'a plus à se soucier puisque LINQ permet de les manipuler de la même façon, comme des objets avec une même syntaxe. Et c'est en cela que LINQ est une réelle innovation. Au même titre que le Framework lui-même permet avec un même EDI, un même langage, une même librairie, de concevoir aussi bien des applications Windows que des applications Web ou des applications pour Smartphone, LINQ avec la même syntaxe permet de trier, de filtrer et de mettre à jour des données qu'il s'agisse d'objets métiers, de fichiers XML ou de sources SQL.

LINQ se décline en plusieurs « arômes » partageant tous une syntaxe de base et l'esprit de LINQ. Mais chacun est spécialisé pour un contexte donné. Pour comprendre la justification et le fonctionnement de ces différentes adaptations de LINQ, suivez le guide…

LINQ to Objects

LINQ to Objects pourrait être vu comme le socle de base. Il a pour but de permettre l'interrogation et la manipulation de collections d'objets. Ces dernières pouvant avoir des *relations hiérarchiques* avec d'autres collections et ainsi former un *graphe*.

Par essence LINQ est donc relationnel, mais pas au sens des bases de données, au sens d'un langage objet.

> *Les bases de données SQL sont fondées sur la notion de tableaux appelés tables, c'est-à-dire des surfaces rectangulaires accessibles en ligne / colonne. Tout est table sous SQL, et un résultat de requête est aussi une table. C'est là toute la puissance de SQL mais c'est aussi sa faiblesse : il ne peut pas retourner un graphe, juste un rectangle avec des cases. LINQ travaille sur des graphes et sait retourner des graphes. SQL ne sait tout simplement pas le faire.*

D'ailleurs LINQ to Objects ne se limite pas à la manipulation des objets créés par le développeur, il permet d'interroger n'importe quelles données, même celles retournées par le système du moment qu'il s'agit d'une collection. Pour s'en convaincre et appréhender tout de suite l'intérêt évident de LINQ prenons un exemple très simple (peu importe pour le moment que la syntaxe vous échappe peut-être) qui consiste à lister les fichiers du répertoire temporaire de Windows.

```
public static void LanceDemo()

{
```

```
  string temp = Path.GetTempPath();

  DirectoryInfo info = new DirectoryInfo(temp);

  var query = from f in info.GetFiles()

              where f.Length > 1024 * 10

              orderby f.Length descending

              select new { f.Length, f.LastWriteTime, f.Name };


  ObjectDumper.Write(query);

  Console.ReadLine();

}
```

La sortie à la console, lorsqu'on appelle la méthode `LanceDemo()` est la suivante :

```
Length=57689462   LastWriteTime=20/11/2012        Name=VSMsiLog26AC.txt
Length=56904858   LastWriteTime=06/08/2013        Name=VSMsiLog6E4B.txt
Length=19757712   LastWriteTime=19/11/2013        Name=VSMsiLog3BA0.txt
Length=18874368   LastWriteTime=06/08/2012        Name=WinSAT_DX.etl
Length=7340032    LastWriteTime=06/08/2012        Name=WinSAT_KernelLog.etl
Length=5891268    LastWriteTime=20/11/2012        Name=dd_WMPPC_5_0_MSI31C5.txt
Length=5882250    LastWriteTime=06/08/2013        Name=dd_WMPPC_5_0_MSI772C.txt
…
Length=11436      LastWriteTime=07/08/2013        Name=SilverlightUI403E.txt
Length=11436      LastWriteTime=06/08/2012        Name=SilverlightUI77FC.txt
Length=11378      LastWriteTime=13/11/2012        Name=SilverlightUI11EE.txt
Length=10996      LastWriteTime=08/11/2013        Name=vs_setup.pdi
```

Lorsque la méthode `LanceDemo()` est invoquée, elle affiche à la console tous les fichiers du chemin temporaire de l'OS en ne prenant que ceux dont la taille est supérieure à 10 Ko, le tout en présentant les fichiers par ordre décroissant de taille. Seule trois informations par fichier sont retournées : sa longueur, sa date de dernière écriture et son nom. Êtes-vous capable de faire un tel traitement aussi clairement et en si peu de lignes sans LINQ ? … Vous connaissez la réponse, c'est non. Et vous commencez donc à comprendre tout l'intérêt de LINQ même avec de simples liste d'objets (et je ne parle pas des graphes d'objets !)…

> *L'affichage est réalisé par ObjectDumper qui est une petite classe outil réalisant tout simplement un foreach pour lister à la console toute collection qui lui est passée. La classe utilise la réflexion pour afficher le nom des propriétés avec leur valeur. Elle n'a rien à voir avec LINQ, c'est une astuce pour afficher rapidement des données. Le code source, provenant de Microsoft, est fourni avec le code des exemples de l'article.*

Construction de la requête

En dehors de l'objet utilitaire d'affichage que vous ne connaissez pas et qui n'a pas d'intérêt dans nos explications, le reste est d'une grande lisibilité : nous obtenons le chemin des fichiers temporaires de Windows puis nous construisons une requête LINQ.

LINQ ressemble beaucoup ici à SQL mais avec certaines spécificités. La première chose et la plus importante c'est qu'il s'agit de code C#, pas d'une chaîne de caractères ne voulant rien dire pour le langage. Il est ainsi contrôlé à la compilation et on dispose d'IntelliSense lors de son écriture. Bien entendu, vous noterez aussi que le plus puissant des SQL ne pourrait rien dans ce cas précis puisque la source est une liste renvoyée par l'OS via le Framework .NET.

Ensuite vous remarquez que par rapport à SQL la requête est en quelque sorte « inversée ». La clause `from` est placée en premier et la clause `select` en dernier…

Pourquoi ce choix qui peut sembler curieux de prime abord ?

L'explication m'a été donnée par Luca Bolognese dans sa conférence sur LINQ aux TechEd 2007 à Barcelone (conférence TLA 308 pour ceux qui y ont participé et qui souhaiterait la voir ou la revoir sur le DVD) : Visual Studio possède une fonction essentielle pour accélérer le développement, *Intellisence*. Et pour que ce dernier puisse fonctionner correctement il faut qu'il sache sur quoi on travaille, ce qui est le cas le plus souvent. Mais dans une requête SQL (ou de type SQL) si on commence par taper le `select` Intellisense sera dans l'impossibilité de fournir une aide concernant les champs disponibles puisque le développeur n'a pas encore tapé la clause `from`… Donc pour permettre à Intellisense d'être utilisable même lors de la frappe des requêtes LINQ il fallait que le nom de la ou des sources de données, le `from`, se trouve en premier. C'est tout bête.

De plus, Luca parle aussi d'expériences menées par Microsoft où des développeurs purement SQL étaient placés devant des écrans pour taper des requêtes, le tout sous l'œil indiscret d'observateurs professionnels cachés derrière des vitres sans tain. Lors

de ces expériences il a pu être noté que le plus souvent un développeur SQL commence par taper la clause `from` pour savoir sur quoi porte sa requête puis il « remonte » pour taper le `select`. LINQ ne ferait donc que reprendre l'ordre réel dans lequel les développeurs saisissent du SQL, même sans en avoir conscience…

Bref, LINQ commence par la clause `from` dans laquelle on nomme une source (et aussi d'éventuelles jointures comme nous le verrons ultérieurement), dans notre exemple la source est fournie par la collection retournée par `GetFiles()` de l'objet `DirectoryInfo`. Dans cette requête on décide d'appeler chaque élément de cette source `f`, au même titre que dans un `foreach` on nomme la variable qui sera renseignée à chaque passage dans la boucle. Il ne faut pas confondre cette notation qui ressemble au premier coup d'œil à un alias de table en SQL (`FROM Customer CLIENTS`). Sous SQL il s'agit bien de l'alias de l'ensemble de données (la table), alors qu'ici il s'agit d'un élément de la collection. A ce stade notre requête possède donc une source de données (la liste des fichiers du répertoire temporaire de l'OS), source qui alimentera à chaque passage une variable appelée `f`. Le type de `f` est déduit de celui de la collection automatiquement (inférence de type comme par le mot clé `var`, voir mon article sur les nouveautés de C# 3.0).

On retrouve ensuite des mots clés classiques de SQL comme la clause `where`. Ici elle nous sert à filtrer sur la taille des fichiers retournés. La clause `orderby` fonctionne comme son homonyme SQL, elle trie les données retournées, ici en ordre descendant avec l'ajout de `descending`.

La clause `select` de l'exemple est peut-être celle qui sera la moins simple à comprendre de prime abord même si son nom est familier en SQL.

En réalité, dans la version la plus simple nous aurions pu écrire `select f ;` c'est-à-dire retourne l'élément `f` de la collection qui répond aux critères de la requête. Dans ce cas la variable `query`, qui contient (virtuellement) le résultat de la requête aurait été une collection d'objets `FileInfo`.

Mais nous ne sommes pas intéressés ici par toutes les informations de `FileInfo`, seuls le nom du fichier, sa taille et sa date de dernière écriture sont utiles pour cette application (choix arbitraire pour la démo, bien sûr). Pourquoi se charger d'objets plus lourds si notre besoin est plus léger ? … LINQ permet de répondre à cette question très facilement : par le mot clé `new` qui va permettre de retourner une instance de tout type et donc, aussi, un type anonyme (*voir mon article sur les nouveautés de C# 3.0*), ce type anonyme, cette nouvelle classe sans nom, sera constituée de trois champs, les trois que nous listons. Nous ne leur donnons pas de

nom car LINQ est assez fin pour les déduire de ceux des propriétés de `f` (une instance de `FileInfo`) … N'indiquant aucun nom, LINQ reprend automatiquement ceux des propriétés de l'objet utilisé (ceux de la classe `FileInfo` donc).

Voilà ce qu'est LINQ to Objects et voilà par un exemple on ne peut plus simple pourquoi cette nouveauté de C# 3.0 va devenir aussi indispensable que tout le reste du langage : tout est données, même une liste de fichiers retournée par l'OS et cette liste peut être filtrée, triée en quelques mots en bénéficiant de IntelliSense…

Un dernier mot. Un peu plus haut je parle de la variable `query` (déclarée par `var`) et j'ajoutais qu'elle contient le résultat de la requête mais « virtuellement ». Pourquoi cette nuance ?

Simplement parce les requêtes LINQ ne sont exécutées qu'au moment où leur résultat est énuméré. Dans notre code cela à lieu dans la boucle de l'objet `ObjectDumper`. Le fait d'écrire une requête LINQ ne déclenche rien. On peut ainsi les regrouper à un même endroit pour plus de lisibilité ou d'autres raisons, elles ne déclencheront leur traitement qu'au moment où on tentera d'énumérer les éléments retournés.

LINQ to ADO.NET

Nous venons de voir que LINQ to Objects est une fonctionnalité de base de C# 3.0 qu'on peut utiliser sur toute collection d'objets (ou graphe d'objets). Nous n'avons utilisé qu'une liste simple dans l'exemple précédent, mais LINQ fonctionne sur des graphes (avec des jointures), c'est-à-dire sur des propriétés qui retournent d'autres collections. Par exemple la fiche d'un client possèdera une propriété « commandes » qui est une collection de toutes les commandes de ce client qui elles-mêmes ont une propriété « lignes » retournant les lignes de chaque commande, etc. LINQ autorisera la navigation dans telles propriétés et sous-propriétés, comme on navigue sur les relations d'une base de données.

L'exemple de LINQ to Objects ne montrait pas cette possibilité car nous allons avoir d'autres occasions de mettre en œuvre la gestion de vrais graphes d'objets, notamment avec LINQ to ADO.NET. Gardez seulement à l'esprit que cela s'applique aussi à LINQ to Objects.

LINQ to ADO.NET, c'est quoi ?

On appelle LINQ to ADO.NET l'ensemble des implémentations de LINQ qui, d'une façon ou d'une autre, doivent accéder à des données relationnelles accessible via les fournisseurs d'accès de ADO.NET.

LINQ to ADO.NET regroupe ainsi trois saveurs différentes de « LINQ »:

LINQ to SQL

Cette version de LINQ gère le mapping entre des types C# et les enregistrements d'une base de données. C'est en quelque sorte le niveau d'entrée de LINQ to ADO.NET.

LINQ to Dataset

Il s'agit de la possibilité d'interroger avec LINQ les données contenues dans un Dataset.

LINQ to Entities

Cette version de LINQ possède un niveau d'abstraction supérieur à LINQ to SQL. Au lieu d'interroger la base de données et d'avoir une sorte d'équivalence entre classes manipulées par LINQ et tables ou vues présentes dans la base de données, LINQ to Entities permet d'écrire des requêtes qui portent sur un *modèle conceptuel*, modèle lui-même mappé sur la base physique. Nous en verrons aussi un exemple car il s'agit de la version de LINQ la plus sophistiquée et la plus puissante.

LINQ to XML

Cette version de LINQ a été adaptée pour permettre la manipulation de données XML. On retrouve ici toute la justification et toute la puissance de LINQ au service des données XML, de plus en plus présentes mais toujours aussi lourdes à exploiter. Grâce à LINQ to XML, et par l'ajout de quelques spécificités permettant de gérer les balises XML, il devient possible d'écrire des requêtes complexes sur des sources XML en quelques instructions et même de construire des fichiers XML de façon intuitive. Loin d'être anecdotique, LINQ to XML révolutionne lui aussi par ses ajouts syntaxiques la façon de travailler avec des données XML.

Les exemples

Le but de cet article est de vous fournir une vision d'ensemble de LINQ et pour cela, plutôt que de rester dans la théorie je souhaite vous montrer du code au travers d'exemples concrets et parlants. En revanche je ne traiterai pas en détail de la syntaxe utilisée, la documentation de Microsoft me semble suffisamment claire pour ne pas en faire une redite.

LINQ to SQL

Si LINQ to Objects est déjà en soi un grand progrès, avec LINQ to SQL on franchit encore une étape qui marquera un tournant dans les langages de programmation, j'en suis convaincu.

En effet, depuis des décennies les développeurs passent une bonne partie de leur temps à tenter de marier deux mondes totalement disjoints et s'éloignant avec le temps : d'une part les langages, d'abord procéduraux, puis simplement « orientés objet » jusqu'au langages comme C# totalement objet, et, d'autre part, les bases de données, d'abord simples fichiers indexés, puis base de données relationnelles de plus en plus sophistiquées. Deux mondes qui ont connu un essor aussi important mais dans deux directions qui ne se sont jamais rejointes.

Les bases de données, comme je le soulignais en début d'article, ne savent travailler que sur un modèle ligne / colonne, avec une entité de base, la table (appelée aussi relation). C'est même leur fondement, la toute première des 12 règles du docteur Codd qui fonda le « modèle relationnel » dans son papier « *A Relational Model of Data for Large Shared Data Banks* » en 1970 (la règle n°1 impose que toute l'information soit présentée uniquement sous la forme de lignes et de colonnes – une table).

Quelles que soient les évolutions des SGBD, l'ajout des champs Blobs, le support des données texte longues, des données XML, voire des « objets », tout cela reste régi par les 12 règles de Codd et ne peut s'en écarter au risque de faire écrouler l'édifice technologique et industriel des SGDB relationnels. D'où la naissance des SGBD objet. Mais si le monde des SGBD relationnels est bien balisé, reposant sur des technologies éprouvées, il n'en va pas de même des SGBD-O. Le premier frein est leur lenteur légendaire, le second est que par manque de standard imposé par les faits, chacune possède un langage d'interrogation plus ou moins ésotérique (API complexes ou variantes exotiques de OQL généralement). Pire, les objets de ces SGBD-O n'ont que très peu de chance de correspondre aux objets des langages utilisés par les développeurs ce qui réclament à nouveau un système de traduction et de mapping… On ne gagne donc pas grand-chose et on se complique surtout la vie (et la maintenance !).

On constate ainsi que malgré quelques tentatives intéressantes les SGBD-O ne rencontrent le plus souvent qu'un succès d'estime comme en ce moment DB4O ou même O2.

Bref, les SGBD-R ont encore de très beaux jours devant eux car ils sont standardisés, rapides, puissants et de nombreux acteurs de ce marché sont implantés depuis suffisamment longtemps pour que chacun puisse faire un choix éclairé et bénéficier de produits rapides, puissants et fiables. SQL Serveur, MySQL, Sybase, DB2 ou Oracle sont des exemples très factuels de ces avantages.

Ce constat serait-il celui d'un statu quo, d'un mariage définitivement impossible entre langage de programmation Objet et SGBD-R ? Comment imaginer marier ces bases de données relationnelles coincées dans leur logique en deux dimensions, ne sachant travailler que sur des rectangles, ne sachant retourner que des rectangles, avec des langages objets eux aussi de plus en plus sophistiqués et travaillant sur des graphes ?

C'est le O/R mapping (ou ORM, *object-relational mapping*), c'est-à-dire un middleware à qui on apprend à la fois comment les données sont stockées dans la base de données et comment elles sont représentées en mémoire par des objets du langage de programmation. A sa charge de créer ces derniers à partir des informations contenu dans la base. L'ORM est une technique intéressante à plus d'un titre puisqu'elle permet au développeur de manipuler des objets en conformité avec la puissance des langages modernes tout en exploitant, pour la persistance, des SGBD-R fiables et largement répandus dans les entreprises.

Le seul problème de l'ORM c'est qu'il n'existe là non plus aucun vrai standard industriel ni même technique, puisque cela va d'une simple couche de type Data Access Layer (DAL) à des Frameworks entiers disposant de leur propre langage d'interrogation, bref, on trouve de tout. Faire un choix est chose ardue. Et quand un développeur ne sait pas comment choisir, il ne choisit pas : il évite et contourne le problème…

Le Framework .NET nous permettait déjà d'écrire des couches d'accès aux données objectivées, les DAL évoqués ci-dessus. Il s'agit d'ailleurs plus d'une sorte de *design pattern* que d'une innovation de la plateforme, c'est une façon d'isoler le code de la source de données très efficace et élégante utilisable, et utilisée, sous d'autres langages.

Pour aller encore plus loin le Framework .NET a proposé les Datasets typés, le mapping est automatisé, mais l'objet final, le Dataset n'est qu'une objectivation

grossière de la source, il continue de présenter des données rectangulaires, pas des graphes d'objets, faisant « remonter » telle une bulle les concepts et la rigidité des SGBD au niveau des objets et du langage. On oblige toujours le développeur et son langage Objet à se plier à la logique des rectangles, le progrès est purement stylistique et pratique : moins de code à saisir (génération automatique de Visual Studio), collections de lignes de données plus simples à manipuler que des requêtes SQL individuelles, typage des informations.

Il fallait donc aller plus loin. Il fallait résoudre le mapping à sa plus simple expression, éviter les redites dans le code, unifier les concepts et surtout les rendre accessibles naturellement, en faire une extension du langage et de la plateforme. LINQ to ADO.NET c'est cela. LINQ to SQL n'est que le premier niveau, celui que nous allons voir maintenant. LINQ to Entities va encore plus loin pour atteindre le niveau d'abstraction le plus élevé possible.

Commençons par le début donc, voici LINQ to SQL.

Dans sa version la plus simple, LINQ to SQL permet de mapper n'importe quelle classe à n'importe quelle table d'une base de données. Nous utiliserons la célèbre base *Northwind* qui, si vous ne la connaissez pas encore, est une base de données classique de type clients / commandes / articles fournie avec SQL Server et utilisée systématiquement dans les démos, rituel auquel je vais faire allégeance…

Les bases du mapping

Je vous propose dans un premier temps de créer une classe `Client` toute simple :

```
public class Client
{
  public string CustomerID { get; set; }
  public string ContactName { get; set; }
  public string Country { get; set; }
}
```

Difficile de faire plus simple… La classe expose trois propriétés, l'identificateur du client, le nom du contact et le pays du client.

> *Nota : les propriétés sont ici déclarées en utilisant une nouveauté de C# 3.0. Cette notation ressemble comme deux gouttes d'eau à celle d'une interface. Les accesseurs (get et set) sont indiqués mais sans code. La ressemblance est frappante mais elle s'arrête là. Nous définissons bien une propriété « normale », sauf que C# créera lui-même le champ privé correspondant (on peut voir les noms de ces champs dans le code IL après compilation). C'est une façon rapide et pratique de créer des propriétés sans, au départ, implémenter de champ privé ou d'accesseurs complexes, tout en laissant la possibilité de faire évoluer le code ultérieurement.*

Comment mapper cette classe sur la table `Customers` de Northwind ?

Il suffit d'utiliser des attributs disponibles par `System.Data.Linq`.

Le premier attribut s'applique à toute la classe et permet d'indiquer à quelle table elle correspond. Le second attribut s'applique à chaque propriété qui possède un pendant dans les colonnes de cette table. Le code de notre classe devient alors :

```
[Table(Name="Customers")]
public class Client
{
  [Column(IsPrimaryKey = true)]
  public string CustomerID { get; set; }
  [Column]
  public string ContactName { get; set; }
  [Column]
  public string Country { get; set; }
 }
```

On peut voir ci-dessus l'attribut `Table` ainsi que l'attribut `Column`. Ces attributs supportent des propriétés ou des constructeurs personnalisés pour affiner leur signification. Ainsi l'attribut `Table` possède une propriété `Name` qui permet d'indiquer le nom physique de la table dans la base de données puisque notre classe possède un nom très différent.

De même l'attribut `Column` supporte, entre autre, la propriété `IsPrimaryKey` pour indiquer que la propriété est la clé primaire de la table, permettant à LINQ de gérer cette contrainte. Comme dans cet exemple j'ai utilisé des noms de propriétés identiques aux noms des colonnes de la table il n'y a rien à ajouter. On pourrait bien entendu faire comme pour la table et préciser le nom des champs dans l'attribut `Column.` Tout cela reste donc simple tout en offrant une certaine souplesse.

Un contexte d'exécution

Pour exploiter notre classe `Client`, nous devons disposer d'un contexte faisant le lien entre cette dernière et la base de données, pour ce faire nous allons dériver notre propre classe de `System.Data.Linq.DataContext` en y ajoutant une propriété pour la table des clients. Cela s'effectue en deux lignes de code :

```
public class DemoContext : DataContext

{

  public Table<Client> Customers;

  public DemoContext(string fileOrServerOrConnection)
                                  :base(fileOrServerOrConnection)  {}

}
```

En créant notre propre classe dérivée de `DataContext` nous pourrons très facilement accéder à la table des clients qui en devient une propriété (`Customers`).

Nous verrons qu'il existe une autre façon de faire qui évite même d'avoir à dériver `DataContext`.

Une fois cela écrit, c'est fort peu vous en conviendrez, nous pouvons travailler sur des objets `Client` provenant de la base de données sans aucun « contact » avec le monde des SGBD !

Première requête

```
public static void LanceDemo()

{ const string northWind = "chaîne de connexion ado.net";

  DemoContext db = new DemoContext(northWind);
```

```
  db.Log = Console.Out;   // affiche le SQL sur la console

  var query =  db.Customers;    // requête ultra simple: toute la table

  ObjectDumper.Write(query);    // affiche le résultat

  Console.ReadLine();           // pause clavier

}
```

Voici un extrait de la sortie console que produit se code :

```
SELECT [t0].[CustomerID], [t0].[ContactName], [t0].[Country]
FROM [Customers] AS [t0]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

CustomerID=ALFKI        ContactName=Maria Anders        Country=Germany
CustomerID=ANATR        ContactName=Ana Trujillo        Country=Mexico
CustomerID=ANTON        ContactName=Antonio Moreno      Country=Mexico
CustomerID=AROUT        ContactName=Thomas Hardy        Country=UK
CustomerID=BERGS        ContactName=Christina Berglund  Country=Sweden
CustomerID=BLAUS        ContactName=Hanna Moos  Country=Germany
CustomerID=BLONP        ContactName=Frédérique Citeaux  Country=France
```

Dans la capture ci-dessus on remarque le texte de la requête SQL réellement envoyée à base de données, c'est une astuce pratique de débogage sur laquelle je vais revenir plus bas. Restons concentrés sur la liste qui suit : on obtient bien la liste de tous les clients de la table `Customers` de la base de données Northwind, chaque fiche possédant bien les trois seules propriétés de la classe `Client` que nous avons vu plus haut. Et c'est tout !

Peut-on rêver ORM plus simple et intégré plus naturellement au langage ?

Certes, pour l'instant nous restons accrochés au modèle physique avec un mapping posant une équivalence entre une classe C# et une table de la base de données. C'est LINQ to SQL. Pour atteindre le niveau supérieur nous devons utiliser LINQ to Entities. Mais pas de précipitation !

Revenons sur le code de l'exemple.

Tout d'abord vous remarquerez la constante qui définit la chaîne de connexion ADO.NET vers la base de données. Cette chaîne peut aussi être issue du fichier de paramétrage de l'application ou de l'utilisateur, bien entendu.

Ensuite nous créons une instance de notre `DataContext` personnalisé, `DemoContext`, auquel nous passons la chaîne de connexion, la variable résultante est `db`.

Petite astuce au passage, comme nous voulons voir pour les besoins de la démo le code SQL qui est réellement envoyé à la base de données, nous faisons pointer la propriété `Log` du `DataContext` vers le flux de sortie de la console. Simple et pratique. Nous verrons plus loin à quel point les requêtes LINQ, tout en restant simples, peuvent générer un SQL bien formé mais complexe que nous n'avons pas, heureusement et grâce à LINQ, à écrire !

La requête que nous utilisons dans cet exemple est la plus simple qu'on puisse concevoir : nous demandons la totalité de la table des clients.

Plus de lignes et de colonnes, plus de tables, plus de données rectangulaires, rien que des instances de la classe `Client`, c'est LINQ qui s'occupe de discuter avec le SGBD dont nous n'avons plus rien à savoir. Plus de couche D.A.L. (*Data Access Layer)* non plus, ni même éventuellement de B.O.L. (*Business Object Layer*) puisque la classe `Client` remplit le rôle de D.A.L. et qu'en y ajoutant un peu de code elle pourrait devenir dans des cas simples en même temps un B.O.L. !

Variante

L'écriture de la classe `DemoContext` dérivée de `DataContext` n'est certes pas bien compliquée… deux lignes de code très utiles puisque notre classe sait ensuite fournir la liste des tables qu'elles connaît (nous aurions pu ajouter de la même façon que les clients, les articles, les commandes etc.).

Mais il est possible de se passer de cette écriture en utilisant les génériques. Dans ce cas, le seul code à concevoir est celui de la classe `Client`… A l'exécution notre code devient le suivant :

```
public static void LanceDemo2()

{

   const string northWind = "chaîne de connexion";

   DataContext db = new DataContext(northWind);
```

```
    Table<Client> Clients = db.GetTable<Client>();

    db.Log = Console.Out;

    var query = Clients;

    ObjectDumper.Write(query);

    Console.ReadLine();
}
```

Dans cette version nous créons directement une instance de `DataContext` sans avoir dérivé cette classe. Bien entendu, `DataContext` ne sait rien de notre table des clients, il n'est donc plus possible de l'utiliser pour obtenir cette dernière. Nous créons ainsi à la volée une variable `Clients` de type `Table<Client>` dont le contenu est récupéré par `GetTable<Client>();`

La requête est ensuite la même. En réalité elle devient superflue… La variable `Clients` contient déjà (potentiellement) tous les clients. Nous pouvons supprimer la variable `query` et passer directement la variable `Clients` à l'objet afficheur :

```
public static void LanceDemo2()
{
    const string northWind = "Data Source=\\sqlexpress;Initial ↵
                             Catalog=Northwind;Integrated Security=True";

    DataContext db = new DataContext(northWind);

    db.Log = Console.Out;

    Table<Client> Clients = db.GetTable<Client>();

    ObjectDumper.Write(Clients);

    Console.ReadLine();
}
```

Vous pourrez d'ailleurs vérifier sur la console que le `DataContext` génère toujours la même requête SQL.

Dans cette version simplifiée nous n'avons fait que décrire la classe `Client` avec deux attributs (`Table` et `Column`). Rien d'autre. Et il nous est possible d'obtenir des clients, de les filtrer, de les trier mais aussi de les modifier. Nous verrons cela un peu plus loin.

Compliquons un peu la requête

Dans l'exemple qui précède nous avons tellement peu utilisé le requêtage de LINQ to SQL que la dernière version du code pouvait même se passer d'une variable pour la requête. Il était intéressant de voir à l'œuvre les mécanismes de base de LINQ, mais le temps est venu d'écrire de vraies requêtes comme nous le ferions en SQL, mais sur des objets avec LINQ.

Nous reprendrons la même structure de code et le même objet `DemoContext` dérivé de `DataContext` exposant la propriété `Customers`.

Filtrage

La première chose qu'on demande à une requête est bien souvent de filtrer les données. Essayons d'obtenir uniquement les clients situés à Londres :

```
public static void LanceDemo3()
        {
            DemoContext db = new DemoContext(northWind);
            db.Log = Console.Out;
            var query = from c in db.Customers
                        where c.Ville == "London"
                        select c;
            ObjectDumper.Write(query);
            Console.ReadLine();
        }
```

Le principe restant le même nous nous concentrerons ici uniquement sur la requête. Elle reste très simple et compréhensible pour qui connaît déjà SQL, et même pour les autres d'ailleurs, pour peu qu'on dispose d'un minimum de vocabulaire anglais.

Si nous regardons la trace SQL (`db.Log`) la requête envoyée à la base de données devient désormais :

```
SELECT [t0].[CustomerID], [t0].[ContactName], [t0].[Country], [t0].[City] AS
[Ville]
FROM [Customers] AS [t0]
WHERE [t0].[City] = @p0
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

On remarque que LINQ to SQL utilise systématiquement des paramètres, ici `@p0`. On peut aussi voir la liste des paramètres et de leurs valeurs sous la requête. La trace `Log` du `DataContext` est très instructive pour qui veut vérifier le SQL transmis à la base de données.

En effet, outre l'intérêt pédagogique que nous exploitons ici, il peut s'avérer dans certains cas nécessaire de proposer sa propre requête, voire le nom d'une procédure stockée pour atteindre les données. LINQ to SQL le gère aussi. Et pour décider quel code sera le plus efficace il s'avère indispensable d'obtenir le code SQL produit par LINQ pour l'inspecter, le `Log` répond à ce besoin.

Tri

Trier des données est tout aussi fréquent que de les filtrer. Voici un exemple de la même requête avec un tri :

```
public static void LanceDemo4()
{
  DemoContext db = new DemoContext(northWind);
  db.Log = Console.Out;
  var query = from c in db.Customers
              where c.Ville == "London"
              orderby c.ContactName ascending
              select new { c.ContactName, c.Ville };
  ObjectDumper.Write(query);
```

```
  Console.ReadLine();

}
```

Ici nous trions les clients retournés par ordre ascendant du nom du contact. Au passage nous ne retournons pas des instances de `Client` mais un type anonyme ne contenant que deux propriétés, le nom du contact et le nom de la ville.

La trace complète à la console donne ceci :

```
SELECT [t0].[ContactName], [t0].[City] AS [Ville]
FROM [Customers] AS [t0]
WHERE [t0].[City] = @p0
ORDER BY [t0].[ContactName]
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

ContactName=Ann Devon   Ville=London
ContactName=Elizabeth Brown     Ville=London
ContactName=Hari Kumar  Ville=London
ContactName=Simon Crowther      Ville=London
ContactName=Thomas Hardy        Ville=London
ContactName=Victoria Ashworth   Ville=London
```

Autre syntaxe

Arrivé à ce stade de l'article je pense que vous commencez à comprendre comment marche LINQ. Mais en réalité ce que nous avons vu n'est qu'une toute petite partie de la syntaxe de LINQ et donc de ses possibilités… Rappelons que ce que vous êtes en train de lire est juste un article présentant LINQ et non un cours sur LINQ. Je vous encourage vivement à consulter la documentation de Microsoft. Et comme je suis taquin et que je ne voudrais surtout pas vous laisser croire qu'il n'y a pas grand-chose d'autre à voir, je vais vous présenter une autre requête LINQ to SQL que je n'expliquerais pas, juste pour créer un peu de frustration et vous donnez l'envie d'aller chercher au-delà de cet article…

```
public static void LanceDemo5()

        {

            DemoContext db = new DemoContext(northWind);

            // db.Log = Console.Out;
```

```
            var query = db.Customers.GroupBy(c => c.Ville,
                                    c => c.Ville+": "+c.ContactName).
                                    Skip(10).Take(8);

            ObjectDumper.Write(query,1);

            Console.ReadLine();

        }
```

Quelques indices malgré tout : la requête prend les 8 premiers clients qui suivent les 10 premiers de la table, une fois celle-ci groupée sur la ville de chaque client. Vous n'avez pas tout compris ? … Je vous l'ai dit, c'est fait exprès !

La trace de sortie est :

```
...
  Bräcke: Maria Larsson
...
  Brandenburg: Philip Cramer
...
  Bruxelles: Catherine Dewey
...
  Buenos Aires: Patricio Simpson
  Buenos Aires: Yvonne Moncada
  Buenos Aires: Sergio Gutiérrez
...
  Butte: Liu Wong
...
  Campinas: André Fonseca
...
  Caracas: Manuel Pereira
...
  Charleroi: Pascale Cartrain
```

Vous retrouverez bien entendu le code de tous les exemples de cet article dans les projets Visual studio fournis dans le même fichier Zip que le PDF de cet article que vous lisez en ce moment (voir sur le site www.e-naxos.com à la rubrique Publications).

Concluons sur LINQ to SQL

L'intention de cet article n'est pas, comme je l'ai déjà mentionné, d'éplucher la syntaxe de LINQ, mais de faire un tour de la technologie avec des exemples pratiques

et simples. Nous n'allons ainsi pas continuer à taper du mapping à la main alors qu'il existe une façon bien plus simple et élégante d'utiliser LINQ avec une base de données, LINQ to Entities. C'est l'objet d'une section à venir, après quelques exemples de LINQ to Dataset et de LINQ to XML.

LINQ to Dataset

LINQ to Dataset est une extension de LINQ permettant de travailler directement sur le contenu de Datasets typés ou non. L'article est fourni avec un projet exemple qui montre quelques manipulations d'un Dataset typé, n'hésitez pas à « jouer » avec.

LINQ to Dataset est tout aussi puissant et versatile que les autres versions de LINQ. Je mettrais juste un bémol à l'utilisation de LINQ sur les Datasets : la bonne pratique (et la RAM de votre machine !) interdit de charger toute une base de données dans un Dataset… Dès lors ce dernier ne doit contenir qu'un sous-ensemble minimal des données de la base, ce qui implique de l'avoir chargé en effectuant déjà un premier filtrage. LINQ to Dataset se place donc après cette première sélection purement SQL pour ne travailler que sur les données en RAM. Cela, à mon sens, restreint bien entendu l'utilité de LINQ ici puisqu'un « bon » Dataset est un Dataset presque vide…

Mais qu'importe, puisqu'on peut se servir de LINQ aussi sur les Datasets, je fais confiance à votre imagination pour en tirer avantage !

Exemple

Reprenons la base de données Northwind et les tables `Customers` et `Orders` dont nous avons tiré un Dataset fortement typé par les mécanismes usuels de Visual Studio. La classe s'appelle `NorthwindDataset` et son schéma est décrit dans le fichier `xsd` accompagné du code source C# produit par VS.

Je passerais sur la façon de créer une instance de ce Dataset et de remplir les deux tables avec les données issues de la base, puisqu'il s'agit ici de procédés déjà connus depuis longtemps de ADO.NET.

Nous en arrivons ainsi à la requête LINQ to Dataset. Celle-ci va utiliser une jointure déclarative entre les clients et les commandes afin de sélectionner tous les clients (ainsi que leurs commandes) dont le nom du contact commence par la lettre A.

```
var query = from c in ds.Customers
            join o in ds.Orders on c.CustomerID equals o.CustomerID
```

```
            into orderlines
            where c.ContactName.StartsWith("A")
            select new
                   {
                    CustID = c.CustomerID,
                    CustName = c.ContactName,
                    Orders = orderlines
                   };
```

Les entités retournées sont formées de l'identificateur du client, de son nom et de la liste de ses commandes. Ce type est anonyme, créé à la volée. On remarquera que nous avons choisi de donner des noms de propriétés différents des noms d'origines, par exemple l'identificateur sera appelé `CustID` et non plus `CustomerID`. De la même façon vous pouvez voir comment l'ensemble des commandes d'un client est « stocké » (virtuellement) dans une variable interne à la requête (`orderlines`) et comment cette grappe d'objets est passée dans l'objet résultat (propriété `Orders`). Tout cela n'a valeur que d'exemple de syntaxe et n'a aucun caractère obligatoire ou fonctionnel bien entendu.

La jointure entre les clients et les commandes est effectuée « à la main » en utilisant `join` d'une façon similaire à la syntaxe SQL de même fonction. Le résultat de cette jointure est nommé (`orderlines`) pour être facilement ré-exploité dans la requête. On notera que la jointure ne fonctionne que sur l'égalité et que pour marquer cette obligation LINQ force l'utilisation du mot `equals` au lieu d'une égalité de type `==`. Il s'agit ici d'éviter toute confusion. Si `==` était autorisé on pourrait être tenté d'utiliser un autre opérateur de comparaison or cela n'aurait pas de sens pour LINQ à cet endroit. En utilisant un mot clé particulier, LINQ marque ainsi une utilisation particulière et spécifique de l'égalité. Il ne s'agit pas d'une interprétation personnelle mais d'une explication donnée par un membre de l'équipe LINQ aux TechEd Microsoft à Barcelone en octobre dernier, c'est donc une vraie info, certes anecdotique, mais de première qualité☺.

LINQ to Dataset est une adaptation de LINQ travaillant en mémoire, de fait il n'y a pas de code SQL généré à visualiser. Je vous fais grâce de la sortie console qui liste

tous les objets, cela ne vous dirait rien. Le projet exemple est de toute façon fourni avec l'article pour que vous puissiez expérimenter la chose par vous-mêmes.

Regardons maintenant comment, au lieu de créer la jointure en LINQ, nous pourrions exploiter la relation existante dans le Dataset typé :

```
var query2 = from c in ds.Customers

             where c.ContactName.StartsWith("A") &&

             c.GetOrdersRows().Any(sv=> sv.ShipVia==2 &&
                                        sv.ShippedDate.Year>1997)

             orderby c.ContactName ascending

             select new { c.ContactName, Orders = c.GetOrdersRows() };
```

Ici la requête est plus complexe. Nous souhaitons obtenir une liste contenant des éléments (type anonyme) constitués d'un nom de contact client et des commandes de ce client. Cela est réglé par la partie `select` de l'instruction.

Toutefois nous souhaitons que seuls les clients dont le nom de contact commence par la lettre A soient sélectionnés. C'est la première partie du `where` qui s'en charge.

Mais pour compliquer les choses nous désirons que seuls les clients ayant au moins une commande passée après l'année 1997 et livrée via le mode 2 (peu importe ce que signifie ce code) soient sélectionnés. Cela est pris en charge par la seconde partie du `where` en utilisant la relation `GetOrdersRows` qui retourne les commandes du client en cours d'analyse par LINQ. Cette méthode a été générée automatiquement par Visual Studio dans le Dataset typé. On remarque l'utilisation du class helper `Any` provenant de LINQ et de ses paramètre sous la forme d'une expression Lambda.

Comme on le voit, LINQ to Dataset est tout aussi sophistiqué que les autres émanations de LINQ. En réalité d'ailleurs nous n'avons rien utilisé qui soit ici spécifique à LINQ to Dataset, les requêtes proposées en exemple auraient pu être effectuée avec LINQ to SQL notamment. Seule la source de données est différente (mapping d'une classe vers une table avec LINQ to SQL ou utilisation des tables d'un Dataset avec LINQ to Dataset). Et c'est bien là la force de LINQ, il en existe des versions spécialisées selon le type de la source de données, mais sa syntaxe et sa puissance ne changent pas.

LINQ to XML

Cette version de LINQ repose sur les mêmes bases que tout ce que nous avons vu jusqu'à maintenant mais elle introduit quelques éléments syntaxique propres. Sa spécificité est de pouvoir travailler sur des sources XML, voire de créer des fichiers XML. Plus de Dataset ni de `DataContext` ici, juste des documents XML, sur disque ou en mémoire.

Il est donc possible avec LINQ to XML d'interroger des documents XML de la même façon qu'un Dataset ou qu'une liste d'objet. Les exemples de syntaxe vus jusqu'ici sont applicables à des sources XML. Toutefois XML utilise un formalisme bien particulier, il existe donc toute une syntaxe LINQ to XML bien spécifique à ce contexte. Ce que nous allons voir maintenant.

L'un des gros avantages de LINQ to XML est de permettre l l'interrogation et la création de documents XML en se passant totalement d'API plus ou moins complexes, et même de XPath, XQuery ou XSLT ! LINQ to XML n'a pas vocation à remplacer ces technologies, il se place juste au-dessus pour donner au développeur un niveau d'abstraction supérieur et simplifier la manipulation des sources XML.

LINQ to XML propose un modèle particulier pour accéder aux données XML, les sources pouvant être un flux (*stream*), un fichier ou du XML en mémoire. En réalité il y très peu de choses à savoir pour utiliser le modèle LINQ to XML, il suffit de connaître les types les plus courants de ce dernier qui sont : `XDocument`, `XElement` et `XAttribute`. Chacun possédant des constructeurs permettant de contrôler les objets créés avec précision. On comprend, bien entendu, rien que par leur nom la vocation de ces classes et leur relation hiérarchique calquant celle du formalisme XML (document / élément / attribut).

Créer des données XML

Regardons d'abord comment il est facile à l'aide des classes indiquées ci-dessus de créer un fichier XML par code :

```
public static void Demo1()
{
  XElement xml = new XElement("clients",
             new XElement("client",
                 new XAttribute("ID", 2),
```

```
                new XElement("société", "e-naxos"),
                new XElement("site", "www.e-naxos.com")),
            new XElement("client",
                new XAttribute("ID", 5),
                new XElement("société", "Microsoft"),
                new XElement("site", "www.Microsoft.com")));
            Console.WriteLine(xml);
            Console.ReadLine();
        }
```

La sortie console de cet exemple sera le contenu de la variable `xml` :

```
<clients>
  <client ID="2">
    <société>e-naxos</société>
    <site>www.e-naxos.com</site>
  </client>
  <client ID="5">
    <société>Microsoft</société>
    <site>www.Microsoft.com</site>
  </client>
</clients>
```

En utilisant `XElement` nous pouvons totalement créer le corps d'un fichier XML, mais pour satisfaire la norme XML ce corps peut être inclus dans un document (ce qui n'est pas indispensable pour le requêtage LINQ). Pour cela on utilise la classe `XDocument`. L'exemple suivant créé un fichier bien formé en exploitant cette possibilité, fichier qui est sauvegardé sur disque dans la foulée.

```
public static void Demo2()
        {
            XDocument doc = new XDocument(
                new XDeclaration("1.0", "utf-8", "yes"),
                new XComment("Liste des clients"),
```

```
                new XElement("clients",
                 new XElement("client",
                     new XAttribute("ID", 2),
                     new XElement("société", "e-naxos"),
                     new XElement("site", "www.e-naxos.com")),
                 new XElement("client",
                     new XAttribute("ID", 5),
                     new XElement("société", "Microsoft"),
                     new XElement("site", "www.Microsoft.com"))));

            doc.Save("ClientsDemo.xml");

            Console.WriteLine(doc);
            Console.ReadLine();
        }
```

Sur disque nous trouvons alors un fichier `ClientsDemo.xml` dont le contenu visionné par le bloc-notes est le suivant :

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--Liste des clients-->
<clients>
  <client ID="2">
    <société>e-naxos</société>
    <site>www.e-naxos.com</site>
  </client>
  <client ID="5">
    <société>Microsoft</société>
    <site>www.Microsoft.com</site>
  </client>
</clients>
```

LINQ to XML ne se limite ainsi pas uniquement à un requêtage simplifié des données XML, il permet aussi d'en créer très facilement.

Marier LINQ to XML avec LINQ to SQL/Dataset/Entities

Produire des données XML est aujourd'hui une activité courante pour une grande partie des applications. Produire ces données à partir d'autres données, notamment celles contenues dans une base SQL ou d'une liste d'objets en mémoire apparaît ainsi d'une grande utilité. Dès lors pourquoi ne pas marier LINQ to XML et sa possibilité de créer simplement des fichiers XML avec LINQ to Dataset, LINQ to SQL ou LINQ to Entities ?

Pourquoi pas en effet… Rien ne s'y oppose. Voyons dans un exemple comment réaliser en quelques lignes ce qui prendrait des pages de code avec d'autres méthodes.

Explications préalables : nous allons utiliser comme source la classe `Client` et le `DataContext` spécialisé de l'exemple de LINQ to SQL étudié plus haut dans cet article. Le code de ces déclarations ne sera pas repris ici pour alléger le texte.

```
public static void Demo3()
{
  const string northWind = "chaîne de connexion";
  using (DemoContext db = new DemoContext(northWind))
  {
    XElement xml = new XElement("clients",
              from c in db.Customers
              where c.ContactName.StartsWith("A")
              orderby c.ContactName
              select new XElement("client",
                     new XAttribute("ID", c.CustomerID),
                     new XAttribute("contact", c.ContactName))
              );
    Console.WriteLine(xml);
    xml.Save("ContactsDemo.xml");
   }
   Console.ReadLine();
```

```
}
```

Ce code créé un corps de document XML à partir des instances de la classe `Client` dont le nom de contact commence par la lettre A, la liste étant triée par ordre alphabétique de ces derniers. Le `select` créé les nouvelles entrées, la table elle-même est créée par le `XElement` de tête s'appelant `xml`. Pour créer un document bien formé et l'enregistrer sur disque par exemple, il suffit d'enchâsser le résultat `xml` dans un `XDocument` et de le sauvegarder par la méthode vu précédemment. L'utilisation d'un `XDocument` n'est en rien indispensable et pour le prouver nous nous en passerons dans cet exemple.

La sortie console de l'exemple est la suivante :

```
<clients>
  <client ID="ROMEY" contact="Alejandra Camino" />
  <client ID="MORGK" contact="Alexander Feuer" />
  <client ID="ANATR" contact="Ana Trujillo" />
  <client ID="TRADH" contact="Anabela Domingues" />
  <client ID="GOURL" contact="André Fonseca" />
  <client ID="EASTC" contact="Ann Devon" />
  <client ID="LAMAI" contact="Annette Roulet" />
  <client ID="ANTON" contact="Antonio Moreno" />
  <client ID="FAMIA" contact="Aria Cruz" />
  <client ID="SPLIR" contact="Art Braunschweiger" />
</clients>
```

Magique non ?

Encore plus intéressant, surtout pour ceux qui, comme moi, restent réfractaires à la logique de Xpath ou XQuery que je n'utilise pas assez souvent pour être à l'aise les quelques fois où j'en aurais besoin, il y a bien entendu la possibilité d'interroger un fichier XML et même de transformer des données (un peu comme XSLT). Par exemple analyser une grappe XML, ne prendre que les éléments qui nous intéressent et créer une liste d'instances de la classe `Client` !

Dans le code qui suit nous prendrons le fichier `ContactsDemo.xml` créé à l'étape précédente et nous allons l'interroger pour créer une liste de string :

```
public static void Demo4()

{
```

```
 XDocument xml = XDocument.Load("ContactsDemo.xml");
 var query = from c in xml.Descendants("client")
             where ((string)c.Attribute("ID")).Contains('M')
             orderby (string)c.Attribute("contact") descending
             select (string)c.Attribute("contact")+" ["+
             (string)c.Attribute("ID")+"]";

 Console.WriteLine("il y a "+query.Count().ToString()+" clients:\n");
 foreach (string s in query) Console.WriteLine(s);
 Console.ReadLine();
}
```

On remarque d'abord qu'en l'absence d'un schéma la requête LINQ to XML fonctionne en utilisant directement les noms des attributs. Sans schéma pas d'aide sur ces derniers qu'il faut taper comme des chaînes et transtyper convenablement.

La sortie console est la suivante :

Le résultat compte 4 clients:

```
Aria Cruz [FAMIA]
Annette Roulet [LAMAI]
Alexander Feuer [MORGK]
Alejandra Camino [ROMEY]
```

Bluffant non ? LINQ to XML C'est un petit pas pour XML mais un pas de géant pour le développeur !

Les grincheux et les impatients diront peut-être qu'on s'éloigne du typage fort qu'offre LINQ puisqu'ici il faut manipuler des balises dont les noms sont saisis en chaînes de caractères (risque d'erreur) et dont les types doivent être obtenus par transtypage (perte du typage fort).

Certes… Mais j'ai une bonne nouvelle pour eux (et pour les autres !), cela s'appelle LINQ to XSD, tout simplement. Il y a déjà eu plusieurs releases en preview de cette technologie qui avait été initiée par le docteur Ralf Lämmel quand il appartenait à l'équipe LINQ to XML. Ce chercheur étant reparti dans sa Germanie natale pour y professer, ce projet à connu quelque retard. Mais très récemment Shyam Pather (Microsoft) à la conférence XML 2007 de Boston a annoncé que tous les efforts seraient faits pour fournir LINQ to XSD, mais dans un second temps, après la sortie de Visual Studio 2008. Encore un peu de patience donc et il sera possible de déférencer les propriétés en bénéficiant du support Intellisense avec LINQ to XSD.

LINQ to Entities

Jusqu'à ce point nous avons pu voir à l'œuvre LINQ sur des collections, sur des données XML, des Datasets et des sources SQL. Ce sont ces dernières qui nous intéressent à nouveau ici.

LINQ to SQL est très puissant et simple à utiliser, mais il s'agit d'une approche centrée sur les données. En réalité LINQ to SQL apparaît dès lors comme une simplification de l'écriture des Data Access Layer en rendant le mapping plus aisé.

Avec LINQ to Entities les données sources ne sont plus des tables ou des objets mappés sur ces dernières, les sources sont purement conceptuelles. Les équipes de LINQ ont réussi à casser le mur entre langage objet et structure de base de données.

Un constat navrant s'impose : une base de données apparaît toujours au développeur sous la forme de son schéma physique alors qu'une base bien conçue a d'abord été murement réfléchie par un développeur spécialisé ou un analyste. Et ce que ces derniers ont modélisé (le mot est lâché) c'est un MCD, un Modèle Conceptuel des Données. Ce MCD a pu être conçu dans un style Merise, ou en notation Entité/Relation, sous le forme de diagrammes UML de classes retranscrits en schémas physiques, peu importe l'outil et la méthodologie utilisée, *une base de données naît d'abord sous les traits d'un modèle conceptuel et non d'un schéma physique*.

La plupart des outils de conception de schémas de base de données savent d'ailleurs produire automatiquement le schéma physique (MPD : Modèle Physique des Données) à partir du MCD, jusqu'au script SQL de création de la base qui n'est qu'un sous-produit de toute cette démarche, le côté « sale », prosaïque, terriblement matériel qui n'intéresse guère de monde à tel point qu'on délègue ce travail à des automates…

Et pourtant ! C'est avec ce schéma physique, cet *avorton* du MPD produit par un automate que le développeur d'applications va devoir se frotter en permanence ! Le rôle du développeur revient ainsi à « défaire » le travail de l'automate pour « remonter » sur le MCD afin de le faire coïncider avec les concepts, objectivés, de son application. Une tâche stupide, semée d'embuches et source d'innombrables erreurs (même si le développeur peut disposer du MCD original de la base de données, cela ne simplifie en rien sa tâche de programmation des accès aux données).

Développeur != Plombier

Les développeurs ne sont pas des plombiers (quelle que soit par ailleurs la noblesse de ce métier) et pourtant ils ont la nette sensation qu'en matière de base de données ils passent leur temps à abouter, tordre et remodeler de la tuyauterie dans l'espoir de la faire coïncider aux besoin de leurs applications. Une gymnastique usante et source d'erreurs entre un espace rectangulaire, celui des SGBD, et la logique des graphes et des objets propre aux langages modernes.

Cette sensation d'être tout le temps en train de « bricoler » est, au-delà du constat technique navrant, quelque chose de peu gratifiant. Et quand un développeur ne se sent pas gratifié par ce qu'il fait il le fait mal.

Changer de dimension

Il faut donc trouver un moyen d'échapper à la logique en 2D des rectangles du docteur Codd.

Lorsqu'on regarde un diagramme de classes d'une application objet bien conçue, on s'aperçoit qu'il n'y a que fort peu de différences avec un MCD de base de données. Au départ ces deux mondes sont finalement très proches ! Le divorce n'est qu'une apparence, un terrible coup du sort : le concepteur de la base est obligé « d'abaisser » le niveau conceptuel de son schéma à celui d'un schéma physique pour satisfaire les exigences du SGBD cible. Le développeur d'applications ne voyant que ce résultat…

Il faut ainsi trouver un moyen pour que la base de données puisse réapparaître sous une forme conceptuelle afin que le développeur ne s'encombre plus des détails et limitations du modèle physique optimisé pour le moteur de base de données et non pour un humain ou une application.

On pourrait supposer créer un outil qui, partant d'un diagramme de classes, produirait d'un côté le schéma physique de la base de données et, de l'autre, le mapping vers les objets en mémoire. En fait cela existe déjà mais n'offre finalement pas la souplesse attendue.

En effet, une base de données répond à des critères normatifs issus des règles de Codd, et rien ne permet d'y échapper. Un bon designer de base de données, même lorsqu'il utilise un outil de conceptualisation, aura toujours tendance à créer des entités et des relations qui bien que modélisant parfaitement les besoins de l'utilisateur resteront empreintes d'une certaine « logique rectangulaire ». Par exemple l'héritage n'existe pas dans les bases SQL, même si certains outils permettent cette notation. Ne parlons pas du support des interfaces, des class helpers, etc… De plus une base de données ne contient que… des données ! Il manque toute la logique des actions (méthodes), l'ordre des séquences, bref tout ce qui fait qu'une application est autrement plus complexe qu'une base de données.

Cette dynamique particulière entre données et actions est le propre de la programmation objet, certainement pas de la création des bases de données ni même de leur modélisation et encore moins de leur optimisation (qui impose souvent de dégrader encore plus le schéma conceptuel et le modèle physique par des processus barbares nommés « dénormalisation »).

De fait, obliger le partage d'un même schéma conceptuel pour la base de données et l'application est une fausse bonne idée. Retour à la case départ.

La solution consiste finalement à laisser les designers de base de données exercer au mieux leur art, et à laisser les développeurs exercer le leur. C'est le principe de LINQ to Entities : à partir du schéma physique de la base de données le développeur va pouvoir créer un modèle conceptuel collant parfaitement aux besoins de son application. Mieux, il pourra créer autant de modèles différents sur un même schéma physique en fonction des concepts qui sont manipulés par telle ou telle autre applications.

LINQ to Entities est ainsi une version de LINQ qui permet d'écrire des requêtes non plus en direction d'une base de données et de son schéma physique, mais bien en direction d'un modèle totalement conceptuel qui va, en quelque sorte, faire écran entre la base de données et les objets de l'application. Charge à LINQ de traduire en SQL tout ce que le développeur fera avec les instances des classes décrites dans le modèle.

Passons à la réalité…

Il était impossible de parler de LINQ sans quelques digressions sur les concepts en jeu tellement ils sont essentiels et lourds de conséquences. Mais dans cet article le moment est venu de passer à un exemple concret !

> *Avertissement : Ce que nous allons voir maintenant est en réalité une extension de Linq to SQL. Dans ce mode on dispose d'un modèle, d'un concepteur visuel spécifique et d'une génération automatiquement de code. Mais en réalité on reste encore en Linq to SQL. Le véritable Linq to Entities repose sur l'Entity Framework. Les nuances visuelles pour cette démo sont faibles, surtout pour un « tour d'horizon », ce qu'est le présent article.*
> *Le véritable Linq to Entities est encore en bêta à l'heure où j'écris ce texte (texte original datant de 2007 !) les différences semblent mineures mais sont essentielles techniquement. Linq to Entities repose sur un mapping XML et non sur des attributs, il permet aussi qu'une entité du modèle soit mappée sur plusieurs tables ou vues ce qui est impossible avec Linq to SQL. La démo qui suit utilise ainsi le mode visuel de Linq to SQL pour faire comprendre ce qu'est Linq to Entities, en raison de cette proximité visuelle et pratique et aussi parce que ce mode de fonctionnement de Linq to SQL est déjà en soi une petite révolution. Mais techniquement, vous l'aurez compris, Linq to Entities et l'Entity Framework vont encore plus loin, même si pour l'instant ce n'est qu'une bêta… (Présentation complète de la version actuelle pour .NET 4.5 sur MSDN : http://msdn.microsoft.com/en-us/library/bb386964.aspx).*

Tout d'abord, LINQ to Entities ne réclame au départ rien de spécial, on peut donc l'utiliser dans tout type de projet. Pour créer le modèle conceptuel, appelé EDM (Entity Data Model, modèle de données des entités), il suffit, au sein d'un projet existant, d'ajouter un nouvel élément et de choisir dans le menu proposé « ADO.NET Entity Model ». L'Entity Framework étant encore en bêta test (voir l'avertissement plus haut) nous utiliserons « LINQ to SQL classes » (voir la figure ci-dessous). Il y a très peu de différences visuelles et nous reviendrons dans un autre article sur les « véritables » EDM de l'Entity Framework. Considérez ici que le principe est le même, la principalement nuance étant que Linq to Entities et l'Entity Framework forment une solution totalement objet autorisant un mapping XML très libre et beaucoup moins limité que celui des classes Linq to SQL de la démo qui suit. Je reviendrai en détail sur l'Entity Framework et Linq to Entities dans un article à venir.

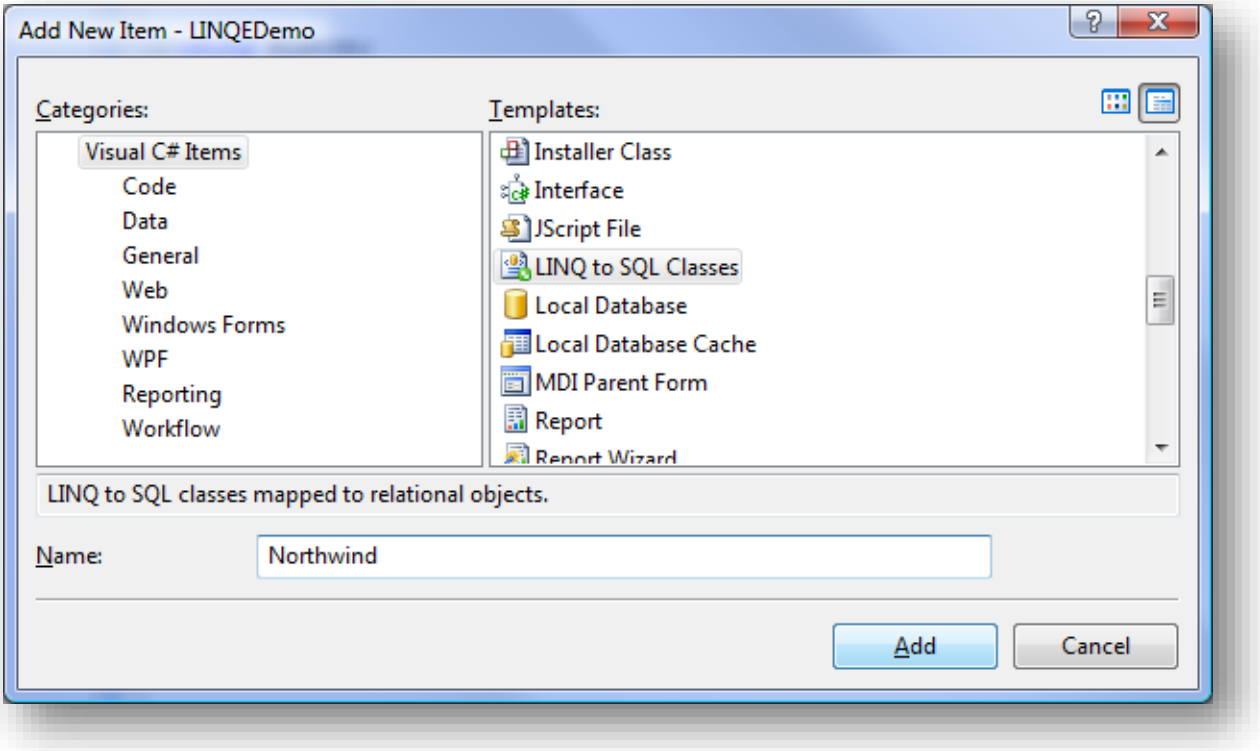


Figure 1 - Création d'un diagramme de classes Linq to SQL

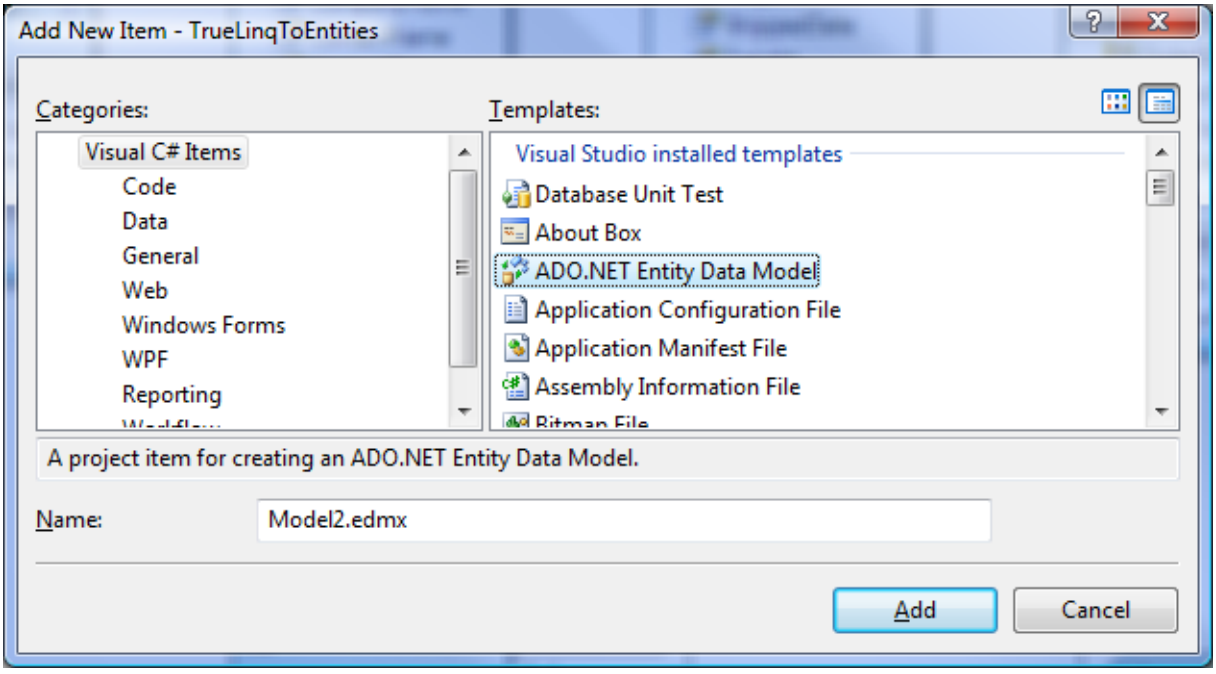


Figure 2 - Création d'un diagramme de classes ADO.NET Entity Data Model

Construire le modèle

La construction d'un modèle s'effectue de façon totalement libre, comme un diagramme de classes UML et avec des outils très proches. Il est donc possible, partant de rien, de concevoir un modèle correspondant exactement au besoin de l'application et dans un second temps de mapper les classes ou propriétés à des tables et champs d'une base de données. Les classes Linq to SQL utilisées pour cette démo n'autorisent qu'un mapping 1:1 (1 classe = 1 table / vue). Linq to Entities et l'Entity Framework permettent eux le mapping multiple démultipliant les possibilités et *renforçant le découplage avec le modèle physique de la base de données*.

Il existe aussi une façon plus simple de procéder que je vais utiliser ici, il suffit de partir d'une connexion avec une base de données et de piocher par drag'n drop les tables qu'on désire adresser puis de modifier le modèle pour le faire tendre vers le niveau d'abstraction de l'application (les modèles ADO.NET Entity models disposent d'un wizard permettant de construire un diagramme depuis une base de données automatiquement) .

En partant de la base Northwind et en sélectionnant les tables Customers et Orders nous arrivons à une première ébauche de modèle. On notera que le designer visuel des classes Linq to SQL dispose d'une toolbox permettant d'ajouter des classes, des associations et des héritages. On retrouve les mêmes fonctions dans les modèles Entity Framework, sauf que le concept de classe est remplacé par celui d'entité.

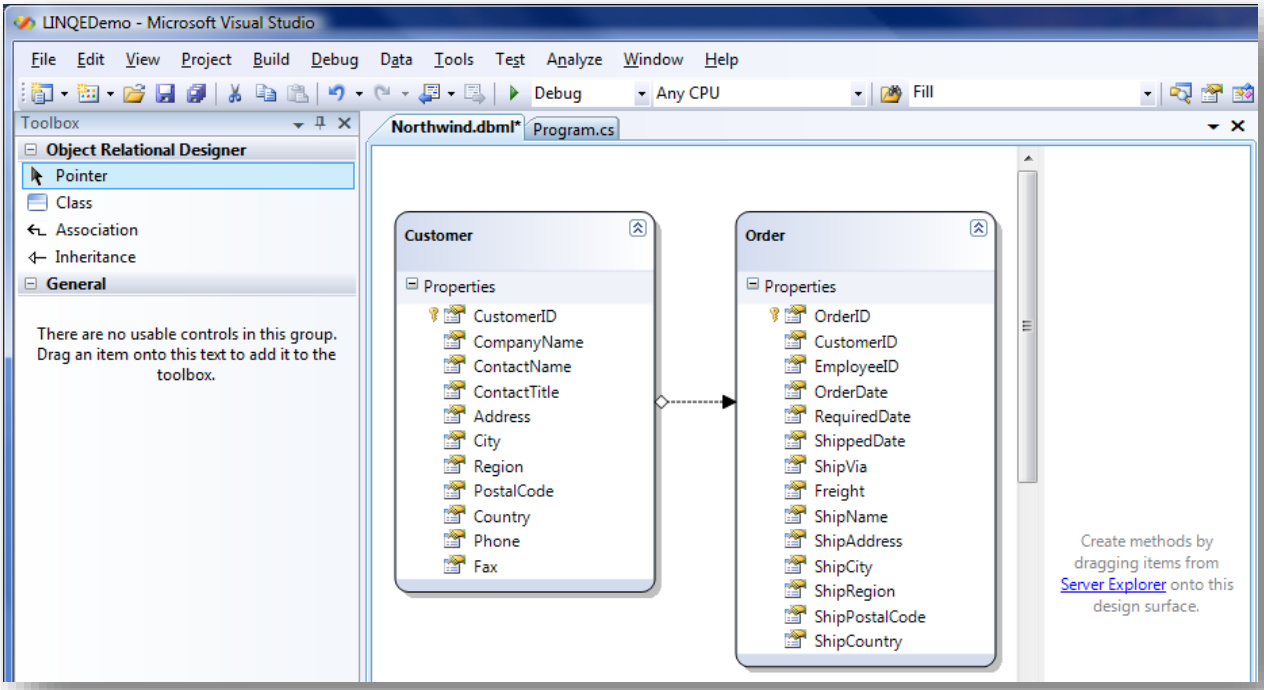


Figure 3 - Un modèle de classes Linq to SQL

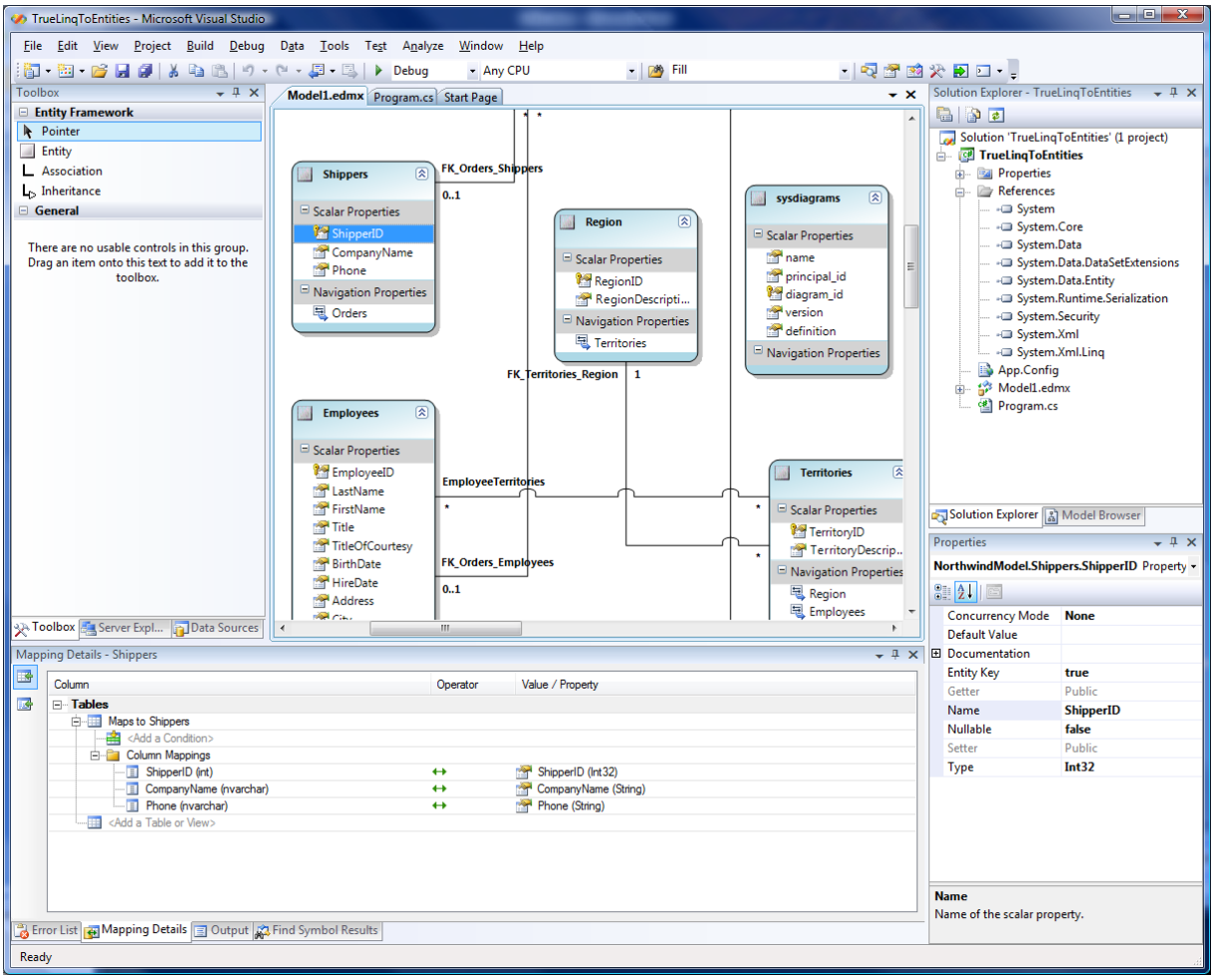


Figure 4 - Un diagramme EDM, Entity Data Model

Le diagramme présenté figure 4 montre le concepteur visuel des EDM. On remarque que si les outils et la présentation sont similaires, on dispose, sous le diagramme, d'un onglet spécifique permettant de gérer le mapping. C'est cet outil-là plus qu'un autre qui, d'un point de vue pratique, change tout puisqu'il permet de manipuler le mapping librement là où les modèles de classes Linq to SQL ne le permettent pas. Au passage vous remarquerez sur le schéma de la figure 4 des relations many-to-many qui ne sont pas prises en charges par les classes Linq to SQL.

A ce stade précis notre modèle de la figure 3 est « brut de fonderie » il calque à 100% le modèle physique puisque nous venons justement de partir de ce dernier. On remarque que par exemple la jointure entre clients et commandes a été automatiquement déduite de celle existante dans la base de données. On remarque aussi que le designer visuel a créé deux classes, enlevant automatiquement au passage le « s » final de Customers et Orders…

Les propriétés de la jointure sont les suivantes :

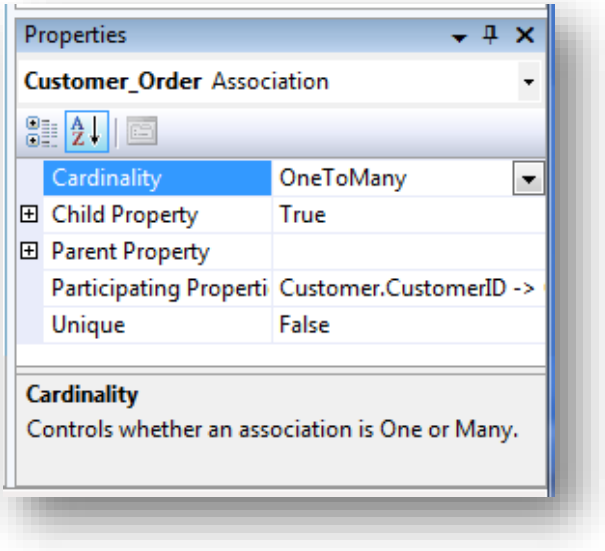


Figure 5 - Propriété d'une relation (Linq to SQL)

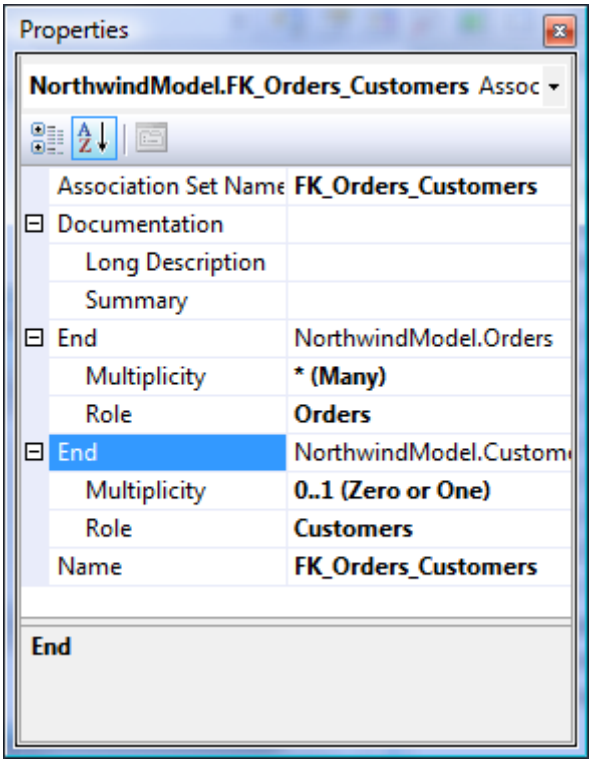


Figure 6 - La même relation exprimée dans un Entity Data Model

Il s'agit ici de définir une relation entre deux classes, comme dans un diagramme de classes UML et non plus une jointure SQL. Notre exemple s'appuie sur Linq to SQL comme indiqué dans l'avertissement d'introduction mais nous continuons à présenter

quand cela est possible, un parallèle avec l'Entity Framework. On remarque ainsi les petites nuances dans la définition d'une relation entre la figure 5, modèle de classes Linq to SQL, et la figure 6, modèle de type Entity Data Model.

Personnaliser le modèle

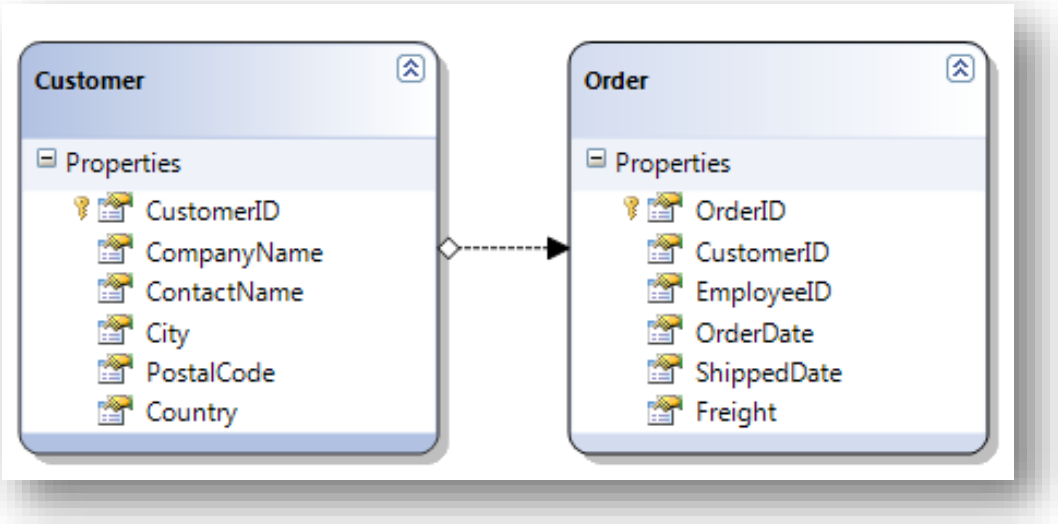
De façon très simple il est possible d'ajouter ou de supprimer des relations, des tables, d'enlever ou d'ajouter des propriétés aux entités, etc…

Il est même possible de modifier le mapping des champs, d'inspecter et de changer le SQL qui sera généré par LINQ et même d'indiquer à ce dernier qu'il faut non pas produire du SQL pour accéder aux données mais utiliser une ou plusieurs procédures stockées (sélection et mise à jour notamment). Les possibilités les plus avancées ne sont disponibles que dans un modèle de type Entity Data Model. Les modèles de classes Linq to SQL restent plus limités en terme de mapping notamment.

LINQ to Entities est une technologie simple mais tellement riche qu'il est bien évidemment hors du cadre du présent article d'en faire le tour complet et en détail. D'autant que cette technologie ainsi que l'Entity Framework ne sont pour l'instant (en 2007) qu'en bêta. Il est plus simple de continuer notre démonstration en se basant sur les modèles de classes Linq to SQL, technologie intégrée à Visual Studio et utilisable tout de suite en production.

Pour illustrer le propos sans entrer dans la complexité d'une véritable application, apportons quelques modifications « symboliques » à notre modèle de classes Linq to SQL en supprimant des propriétés et en complémentant les entités du modèle par du code personnalisé (on pourrait aller encore plus loin dans un modèle EDM).

En premier lieu nous ne conservons que quelques informations pour chaque entité, ce qui produit le diagramme suivant :



Dans un second temps nous allons ajouter du code à chaque entité. Un clic droit et « voir le code » affiche l'éditeur de code placé sur la définition de l'entité. On remarque que tout le code automatique est caché dans un autre fichier et que les classes présentées sont vides (classes partielles). Pour faire simple modifions les méthodes `ToString()` de chaque entité :

```
namespace LINQEDemo
{
    partial class Customer
    {
        public override string ToString()
        {
            return "Société "+CompanyName;
        }
    }

    partial class Order
    {
        public override string ToString()
        {
```

```
            return "Cde n° " + OrderID + " Pour: " + Customer.CompanyName;
        }
    }
```

On imagine bien qu'il est possible d'ajouter tout le code fonctionnel dont on a besoin, on est en C# dans du code « normal » et bénéficiant de l'ensemble des possibilités du Framework. Nous nous arrêterons ici pour les personnalisations.

Premier test du modèle

Pour tester notre modèle et ses personnalisations, le plus simple est de demander une fiche client et une fiche commande, isolément, et d'en faire un affichage en chaînes de caractères (ce qui appellera les méthodes `ToString` surchargées) :

```
public static void Demo1()
{
  using (NorthwindDataContext db = new NorthwindDataContext())
  {
    Console.WriteLine(db.Orders.First());
    Console.WriteLine(db.Customers.First());
    Console.ReadLine();
  }
}
```

La sortie console nous montre alors :

```
Cde n° 10248 Pour: Vins et alcools Chevalier
Société Alfreds Futterkiste
```

La première ligne affiche la première commande de la base de données mise en forme par notre surcharge de `ToString()`. Il en va de même pour le premier client.

Au passage on remarque que la technique est similaire à LINQ to SQL puisque nous utilisons un `DataContext` personnalisé. La seule différence est que ce dernier est généré automatiquement par le designer de Visual Studio. C'est bien parce que nous sommes toujours sous LINQ to SQL… la véritable révolution se trouve dans l'Entity Framework et Linq to Entities que avons survolé ici. Sous Entity Framework, un EDM donne naissance à un `ObjectContext` et non plus un `DataContext`. La différence semble mineure, mais elle est techniquement profonde et ouvre des possibilités bien plus grandes que celles de Linq to SQL.

Ajout d'une procédure stockée au modèle

Par drag'n drop il est possible de déposer sur un modèle de classes Linq to SQL des procédures stockées depuis le gestionnaire de serveurs vers un espace vertical collé au modèle. Cela ajoute la PS à ce dernier en tant que méthode du `DataContext` personnalisé… Ajoutons ainsi la PS `Ten_Most_Expensive_Products` et interrogeons-là pour tester le fonctionnement de l'ensemble :

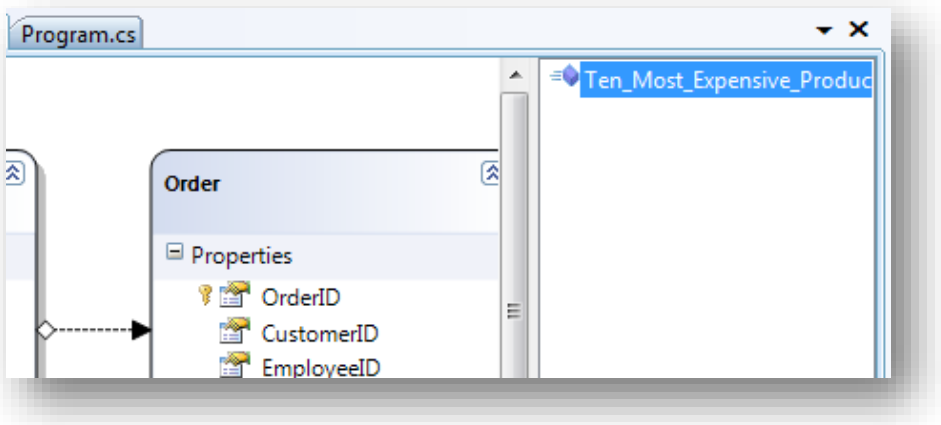


Figure 7 - L'ajout d'une procédure stockée à un modèle

On voit sur l'image ci-dessus la procédure stockée ajoutée dans la colonne de droite, collée au modèle (les procédures stockées sont gérées un peu différemment sous Entity Framework).

Le code pour activer cette méthode est on ne peut plus simple (la classe `ObjectDumper` est la même que celle utilisée plus avant dans cet article) :

```
using (NorthwindDataContext db = new NorthwindDataContext())

{

  ObjectDumper.Write(db.Ten_Most_Expensive_Products());
```

```
  Console.ReadLine();

}
```

Code qui produit la sortie console suivante :

```
TenMostExpensiveProducts=Côte de Blaye  UnitPrice=263,5000
TenMostExpensiveProducts=Thüringer Rostbratwurst        UnitPrice=123,7900
TenMostExpensiveProducts=Mishi Kobe Niku        UnitPrice=97,0000
TenMostExpensiveProducts=Sir Rodney's Marmalade         UnitPrice=81,0000
TenMostExpensiveProducts=Carnarvon Tigers       UnitPrice=62,5000
TenMostExpensiveProducts=Raclette Courdavault   UnitPrice=55,0000
TenMostExpensiveProducts=Manjimup Dried Apples  UnitPrice=53,0000
TenMostExpensiveProducts=Tarte au sucre         UnitPrice=49,3000
TenMostExpensiveProducts=Ipoh Coffee    UnitPrice=46,0000
TenMostExpensiveProducts=Rössle Sauerkraut      UnitPrice=45,6000
```

On peut de la même façon ajouter et gérer des PS qui attendent des paramètres. C'est le cas dans la base de données Northwind de la procédure `CustOrderHist` qui, une fois ajoutée au modèle, donnera naissance à une méthode de même nom attendant en paramètre un ID client.

```
using (NorthwindDataContext db = new NorthwindDataContext())

{

  db.Log = Console.Out;

  ObjectDumper.Write(db.CustOrderHist("ALFKI"));

  Console.ReadLine();

}
```

Nous avons ajouté ici la sortie de la trace SQL sur la console. De fait, pour le client indiqué la sortie console est la suivante (avec en premier le SQL produit par LINQ) :

```
EXEC @RETURN_VALUE = [dbo].[CustOrderHist] @CustomerID = @p0
-- @p0: Input NChar (Size = 5; Prec = 0; Scale = 0) [ALFKI]
-- @RETURN_VALUE: Output Int (Size = 0; Prec = 0; Scale = 0) [Null]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8


ProductName=Aniseed Syrup       Total=6
ProductName=Chartreuse verte    Total=21
ProductName=Escargots de Bourgogne      Total=40
ProductName=Flotemysost         Total=20
```

```
ProductName=Grandma's Boysenberry Spread        Total=16
ProductName=Lakkalikööri        Total=15
ProductName=Original Frankfurter grüne Soße     Total=2
ProductName=Raclette Courdavault        Total=15
ProductName=Rössle Sauerkraut   Total=17
ProductName=Spegesild   Total=2
ProductName=Vegie-spread        Total=20
```

Nous disposons ainsi de l'historique des commandes d'un client sous la forme d'une simple méthode du DataContext et notre application peut utiliser directement le résultat sous la forme d'une collection typée.

Utiliser les relations

Il est temps d'interroger le modèle en exploitant les relations qu'il définit. Voici une requête un petit plus complexe :

```
public static void Demo4()
{
   using (NorthwindDataContext db = new NorthwindDataContext())
   {
     db.Log = Console.Out;
     var query = (from c in db.Customers
                  select new
                  {
                    c.CompanyName,
                    c.City,
                    Orders = from o in c.Orders
                             where o.ShipCode == 2
                             select new { o.OrderID, o.ShipCode }
                  }).Take(5);

     ObjectDumper.Write(query,2);
     Console.ReadLine();
```

```
    }
}
```

Explications : nous demandons ici à LINQ de nous fournir une liste constituée d'éléments dont le type, anonyme, est formé de trois propriétés : le nom de la société du client, la ville du client et une propriété `Orders` qui est elle-même une liste. Cette liste est constituée elle-aussi d'éléments de type anonyme possédant deux propriétés, l'ID de la commande et le mode de livraison. La propriété `Orders` est fabriquée sur la base d'une sous-requête LINQ filtrant les commandes de chaque client et ne retenant que celles ayant un mode de livraison égal au code 2. Enfin, nous ne prenons que les 5 premiers clients de la liste globale (par l'opération `Take(5)` appliquée à toute la requête mise entre parenthèses).

Je vous fais grâce de la sortie console qui montre outre le résultat attendu les diverses requêtes SQL envoyées à la base de données par LINQ (grâce à `db.Log` redirigée vers la console).

Créer des relations

Si nous ajoutons la table `Suppliers` (fournisseurs) à notre modèle nous disposons d'une entité de plus dans ce dernier mais aucune relation n'est déduite de la base de données. En effet, il n'existe aucune jointure entre commandes et fournisseurs ni entre ces derniers les clients. Il existe bien entendu dans la base de données un chemin reliant toutes ces entités en passant par les lignes de commandes et les articles. Mais d'autres chemins sont envisageables.

De plus, nous travaillons sur un modèle conceptuel propre à notre application, nous ne sommes que très peu intéressés par la réalité de l'organisation de la base de données. Notamment, nous souhaitons ici obtenir une liste contenant pour chaque client la liste des fournisseurs se trouvant dans la même ville. Cela n'a pas de sens pour la base de données telle qu'elle a été conçue, mais cela en a un pour nous, ponctuellement dans notre application (par exemple étudier comment faire livrer les clients directement par les fournisseurs les plus proches au lieu de faire transiter la marchandise par notre stock…).

Nous allons pour cela créer une relation « à la volée » dans une requête :

```
using (NorthwindDataContext db = new NorthwindDataContext())
```

```
{
  var query = from c in db.Customers
              join s in db.Suppliers on
              c.City equals s.City into sups
              where sups.Count() > 0
              select new { c.CompanyName, c.City, Suppliers = sups };
  ObjectDumper.Write(query, 2);
  Console.ReadLine();
}
```

La sortie console ne vous ferait rien voir de plus, mais regardons le code SQL généré et méditez sur la différence de lisibilité entre les quelques lignes LINQ ci-dessus et ce pavé SQL ci :

```
SELECT [t0].[CompanyName], [t0].[City], [t1].[SupplierID], [t1].[CompanyName] AS
 [CompanyName2], [t1].[ContactName], [t1].[ContactTitle], [t1].[Address], [t1].[
City] AS [City2], [t1].[Region], [t1].[PostalCode], [t1].[Country], [t1].[Phone]
, [t1].[Fax], [t1].[HomePage], (
    SELECT COUNT(*)
    FROM [dbo].[Suppliers] AS [t3]
    WHERE [t0].[City] = [t3].[City]
    ) AS [value]
FROM [dbo].[Customers] AS [t0]
LEFT OUTER JOIN [dbo].[Suppliers] AS [t1] ON [t0].[City] = [t1].[City]
WHERE ((
    SELECT COUNT(*)
    FROM [dbo].[Suppliers] AS [t2]
    WHERE [t0].[City] = [t2].[City]
    )) > @p0
```

```
ORDER BY [t0].[CustomerID], [t1].[SupplierID]

-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [0]

-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

Je vous laisse faire vos propres déductions…

Modifications des données

Interroger de façon simple et contrôlée les données est une chose, que LINQ to Entities fait fort bien, mais qu'en est-il de la modification des données ?

Si vous vous attendez à des tonnes de code vous allez être déçus !

Modifions une fiche client pour commencer :

```
public static void Demo6()
{
  using (NorthwindDataContext db = new NorthwindDataContext())
  {
    var cust = db.Customers.First(c => c.CustomerID == "PARIS");
    cust.PostalCode = "75016";
    db.SubmitChanges();
  }
}
```

Nous obtenons le client dont l'ID est « PARIS », nous modifions son code postal puis nous sauvegardons les modifications dans la base de données… Plus simple je n'ai pas en stock. En revanche je peux vous montrer le code SQL généré par LINQ pour ces trois petites lignes de C# :

```
SELECT TOP (1) [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[City], [t0].[PostalCode], [t0].[Country]
FROM [dbo].[Customers] AS [t0]
```

```
WHERE [t0].[CustomerID] = @p0
-- @p0: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [PARIS]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

UPDATE [dbo].[Customers]
SET [PostalCode] = @p6
WHERE ([CustomerID] = @p0) AND ([CompanyName] = @p1) AND ([ContactName] = @p2)
AND ([City] = @p3) AND ([PostalCode] = @p4) AND ([Country] = @p5)
-- @p0: Input NChar (Size = 5; Prec = 0; Scale = 0) [PARIS]
-- @p1: Input NVarChar (Size = 17; Prec = 0; Scale = 0) [Paris spécialités]
-- @p2: Input NVarChar (Size = 14; Prec = 0; Scale = 0) [Marie Bertrand]
-- @p3: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [Paris]
-- @p4: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [75012]
-- @p5: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [France]
-- @p6: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [75016]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

Personnellement j'ai une petite préférence pour la version LINQ. Pas vous ?

Il est bien entendu possible de continuer comme ça très longtemps avant d'avoir fait le tour de LINQ. Par exemple ajouter une commande à un client (il suffit de créer un objet `Order`, de l'initialiser, de l'ajouter par `Customer.Add()` à l'objet client et de sauvegarder les changements…).

LINQ to Entities propose des mécanismes autrement sophistiqués notamment dans le tracking des modifications faites aux objets et à l'identification de ces derniers. Car bien entendu les grappes ou les objets isolés que nous récupérons n'apparaissent que comme de simples instances pour le développeur, mais au moment de mettre à jour les données comment LINQ sait-il que ces objets proviennent bien de la base de données, que le client 1256 dont on a changé l'ID (cela arrive) est bien le 1256 dans la base (alors que nous n'avons pas nous-mêmes conservé l'ancienne valeur) ? etc, etc…

D'autres questions du même type se posent lorsqu'il s'agit d'envoyer un objet en dehors de la machine (par du remoting par exemple) puis de le récupérer modifié et d'appliquer les changements à la base de données. On pourra aussi se demander comment les transactions sont gérées par LINQ.

Tout cela a bien entendu des réponses, mais nous dépasserions largement les limites de cet article de présentation dont les proportions ont explosé au fil de ce voyage que je voyais un petit plus court au départ…

N'hésitez pas à consulter mon blog, certains billets répondent aux questions soulevées ici et d'autres réponses viendront s'ajouter avec le temps comme, bien entendu, un article totalement consacré à Linq to Entities et l'Entity Framework.

Conclusion

J'ai bien cru ne jamais atteindre ce mot magique ! Conclusion.

Il est effectivement temps de conclure. LINQ est une déesse polymorphe dont on ne saurait se lasser de décrire les courbes gracieuses, mais il n'est de bonne compagnie qui ne se quitte, alors je vous dis à bientôt pour d'autres articles qui visiterons plus en détail d'autres aspects de Visual Studio et du Framework .NET…

Class Helper, LINQ et fichiers CSV

C# 3.0 a apporté des nouveautés syntaxiques qui ne se bornent pas à être seulement des petits suppléments qu'on peut ignorer, au contraire il s'agit de modifications de premier plan qui impactent en profondeur la façon d'écrire du code si on sait tirer parti de ces nouveautés. J'en ai souvent parlé, au travers de billets de ce blog ou d'articles comme celui décrivant justement les nouveautés syntaxiques de C#3.0. Je ne vais donc pas faire de redites, je suppose que vous connaissez maintenant tous ces ajouts au langage. Ce que je veux vous montrer c'est qu'en utilisant correctement C# 3.0 on peut écrire un code incroyable concis et clair pour résoudre les problèmes qui se posent au quotidien au développeur.

Le problème à résoudre

Vous recevez un fichier CSV, disons le résultat d'une exportation depuis un soft XY des ventes à l'export de votre société. Vous devez rapidement ajouter la possibilité d'importer ces données dans l'une de vos applications mais après les avoir interprétées, triées, voire filtrées.

La méthode la plus simple

Rappel : un fichier CSV est formé de lignes comportant plusieurs champs séparés par des virgules. Les champs peuvent être délimités par des double quotes.

A partir de cette description voyons comment avec une requête LINQ nous pouvons lire les données, les filtrer, les trier et les mettre en forme. Le but ici sera de générer en sortie une chaîne par enregistrement, chaîne contenant des champs "paddés" par

des espaces pour créer un colonnage fixe. On tiendra compte des lignes débutant par le symbole dièse qui sont considérées comme des commentaires.

```
 1:  string[] lines = File.ReadAllLines("Export Sales Info-demo.csv");
 2:  var t1 = lines
 3:      .Where(l => !l.StartsWith("#"))
 4:      .Select(l => l.Split(','))
 5:      .OrderBy(items=>items[0])
 6:      .Select(items => String.Format("{0}{1}{2}",
 7:          items[1].PadRight(15),
 8:          items[2].PadRight(30),
 9:          items[3].PadRight(10)));
10:  var t2 = t1
11:      .Select(l => l.ToUpper());
12:  foreach (var t in t2.Take(5))
13:      Console.WriteLine(t);
```

La sortie (avec le fichier exemple fourni) sera :

```
SAN JOSE        CITY CENTER LODGE              CA
SAN FRANCISCO   KWIK-E-MART                    CA
SEATTLE         LITTLE CORNER SWEETS           WA
SEATTLE         LITTLE CORNER SWEETS           WA
IRVINE          PENNY TREE FOODS CORPORATION   CA
```

Cette méthode, très simple, ne réclame rien d'autre que le code ci-dessus. La solution est applicable à tout moment et s'adapte facilement à toute sorte de fichiers sources. Elle possède malgré tout quelques faiblesses. Par exemple les champs contenant des doubles quotes ou les champs mal formés risquent de faire échouer la séquence. Dans certains cas cela sera inacceptable, dans d'autres on pourra parfaitement protéger la séquence dans un bloc try/catch et rejeter les fichiers mal formés. Une fois encore il ne s'agit pas ici de montrer un code parfaitement adapté à un problème précis, mais bien de montrer l'esprit, la façon d'utiliser C# 3.0 pour résoudre des problèmes courants.

Expliquons un peu ce code :

La ligne 1 charge la totalité du fichier CSV dans un tableau de strings. La méthode peut sembler "brutale" mais en réalité elle est souvent très acceptable car de tels fichiers dépassent rarement les quelques dizaines ou centaines de Ko et la RAM de nos machines modernes n'impose en rien une lecture bufferisée, tout peut tenir d'un bloc en mémoire sans le moindre souci. Cela nous arrange ici il faut l'avouer mais l'utilisation d'un flux bufferisé resterait parfaitement possible.

Nous disposons maintenant d'un tableau de chaînes, chacune étant formatée en CSV. La première requête LINQ (variable "t1" en ligne 2) fait l'essentiel du travail :

- gestion des commentaires (ligne 3)
- décomposition des champs (ligne 4)
- tri sur l'un des champs (ligne 5)
- génération d'une sortie formatée (lignes 6 à 9)

Ce qui est merveilleux ici c'est que nous avons en quelques lignes un concentré de ce qu'est une application informatique : acquisition de données, traitement de ces données, production de sorties exploitables par un être humain ou une autre application. En si peu de lignes nous avons réalisé ce qui aurait nécessité une application entière avec les langages de la génération précédente comme C++ ou Delphi. C'est bien ce bond en avant que représente C# 3.0 qui est ici le vrai sujet du billet.

Une méthode plus complète

La séquence que nous avons vue plus haut répond au problème posé. Elle possède quelques lacunes. Celles liées à sa trop grande simplicité (certains cas du parsing CSV ne sont pas correctement pris en compte) et celles liées à sa forme : c'est un bout de code qu'il faudra copier/coller pour le réutiliser et qui viendra "polluer" nos requêtes LINQ les rendant plus difficiles à lire.

Si ces lacunes sont acceptables dans certains cas (règlement ponctuel d'un problème ponctuel) dans d'autres cas on préfèrera une approche plus facilement réutilisable. Notamment si nous sommes amenés à traiter plus ou moins souvent des fichiers CSV nous avons intérêt à encapsuler le plus possible le parsing et à le rendre plus facilement reexploitable.

L'une des voies serait de créer une classe totalement dédiée à cette tâche. C'est une solution envisageable mais elle est assez lourde et amène son lot de difficultés.

Nous allons choisir ici une autre approche, celle de la création d'un class helper. C'est à dire que nous souhaitons non pas créer une classe qui traite un fichier CSV comme un tout, mais nous voulons pouvoir parser n'importe quelle chaîne de caractères formatée en CSV. L'approche est très différente. Dans le premier cas il nous faudra complexifier de plus en plus notre classe, voire créer une hiérarchie de classes pour traiter les fichiers CSV mais aussi les flux mémoire CSV, et puis encore les services Web retournant du CSV, etc, etc.

Dans le second cas nous allons plutôt ajouter la capacité à la classe string de parser une chaîne donnée. La source de la chaîne ne nous importe pas. Il pourra s'agir d'un élément d'un tableau de chaîne comme dans le premier exemple autant que d'une chaîne lue depuis un flux mémoire, une section data d'un fichier XML, etc. D'un certain sens nous acceptons d'être moins "complet" que l'option "classe dédiée CSV", mais nous gagnons en agilité et en "réutilisabilité" en faisant abstraction de la provenance de la chaîne à parser. Bien entendu nous pouvons nous offrir le luxe d'une telle approche car nous savons que nous pouvons nous reposer sur C# 3.0, ses class helpers et LINQ...

Le projet qui accompagne ce billet contient tout le code nécessaire et même un fichier CSV d'exemple, nous n'entrerons pas dans les détails de l'implémentation du class helper lui-même qui étend la classe string, parser du CSV n'est qu'un prétexte sans plus d'intérêt dans ce billet.

La façon d'écrire un class helper est décrite dans mon article sur C# 3.0, regardons juste sa déclaration :

```
public static string[] CsvSplit(this String source)
```

Cette méthode est déclarée au sein d'une classe statique de type "boîte à outils" pouvant centraliser d'autres méthodes utilitaires. La déclaration nous montre que le class helper s'applique à la classe String uniquement (this String source) et qu'elle retourne un tableau de chaîne (string[]).

Une fois le class helper défini (code complet dans le projet accompagnant l'article) il nous est possible d'écrire des requêtes LINQ du même type que celle du premier exemple. Mais cette fois-ci le parsing CSV est réalisé par le class helper ce qui permet à la fois de le rendre plus sophistiqué et surtout de ne plus avoir à le copier/coller dans les requêtes LINQ...

Voici un exemple d'utilisation du class helper sur le même fichier CSV. Ici nous parsons la source, nous la trions, nous éliminons les lignes de commentaire mais aussi nous créons en réponse une classe anonyme contenant le nom du contact, sa ville et le nom de sa société. Il est dès lors possible de traiter la liste d'objets résultat comme n'importe quelle liste : affichage, traitement, génération d'état, etc.

```
 1:  var data = File.ReadAllLines(
             "Export Sales Info-demo.csv").Where(s=>!s.StartsWith("#"))
 2:      .Select(l =>
 3:  {
 4:  var split = l.CsvSplit();
 5:  return new
 6:              {
 7:                  Contact = split[0],
 8:                  City = split[1],
 9:                  Company = split[2]
10:              };
11:  }  ).OrderBy(x=>x.Contact);
12:
13:  foreach (var d in data.Take(5))
14:      Console.WriteLine(d);
```

Ce qui, avec le même fichier source, affichera à la console :

```
{ Contact = Allen James, City = San Jose, Company = City Center Lodge }
{ Contact = Bart H. Perryman, City = San Francisco, Company = Kwik-e-mart }
{ Contact = Beth Munin, City = Seattle, Company = Little Corner Sweets }
{ Contact = Beth Munin, City = Seattle, Company = Little Corner Sweets }
{ Contact = Bruce Calaway, City = Irvine, Company = Penny Tree Foods Corporation }
```

Conclusion

La bonne utilisation de C# 3.0 permet de réduire significativement la taille du code donc de réduire dans les mêmes proportions les bugs et d'augmenter la productivité. Apprendre à utiliser cette nouvelle approche c'est gagner sur tous les plans...

LINQ to {tapez ce que vous voulez ici}. Des providers à foison !

Vous connaissez LINQ to Object, LINQ to Dataset, LINQ to XML, etc... Microsoft livre déjà une large palette de fournisseurs de données (*providers*) pour LINQ. Dans sa grande sagesse Microsoft a aussi publié les spécifications (et des exemples) permettant à tous de développer des fournisseurs de données pouvant fonctionner avec LINQ. Cette ouverture extraordinaire permet d'interroger avec LINQ des données de toute sorte sans avoir besoin de les transformer au préalable.

Mais savez-vous qu'à ce jour **il existe déjà plus d'une trentaine de providers LINQ** ? Et que forcément certains vous rendrons des services inestimables, à condition d'en connaître l'existence... Vous pouvez même écrire *vos propres providers LINQ* ! Je vous conseille d'ailleurs cette page du Wayward blog qui propose une série de billets expliquant comment créer un tel provider.

Un exemple parmi tant d'autres pour illustrer le propos : *LINQ To Amazon*. Il permet tout simplement d'interroger directement Amazon pour sélectionner des livres selon autant de critères que nécessaire, et sans autre programmation qu'une requête LINQ. Fantastique non ? Un petit exemple pour concrétiser la chose :

```
var query =
  from book in new Amazon.BookSearch()
  where
    book.Title.Contains("ajax") &&
    (book.Publisher == "Manning") &&
    (book.Price <= 25) &&
    (book.Condition == BookCondition.New)
  select book;
```

Cool non ? LINQ to Amazon est présenté dans le livre LINQ in Action (que je vous conseille au passage) et vous pouvez accéder au blog de Fabrice qui le présente.

Mais cela n'est qu'un exemple, et il existe bien d'autres fournisseurs de données LINQ, en voici une liste (avec liens - shift clic pour les ouvrir dans une nouvelle fenêtre) :

- LINQ to Amazon
- LINQ to Active Directory
- LINQ over C# project

- LINQ to Continuous Data (CLinq)
- LINQ to CRM
- LINQ To Geo - Language Integrated Query for Geospatial Data
- LINQ to Excel
- LINQ to Expressions (MetaLinq)
- LINQ Extender (Toolkit for building LINQ Providers)
- LINQ to Flickr
- LINQ to Google
- LINQ to Indexes (LINQ and i40)
- LINQ to IQueryable (Matt Warren on Providers)
- LINQ to JSON
- LINQ to LDAP
- LINQ to NHibernate
- LINQ to JavaScript
- LINQ to LLBLGen Pro
- LINQ to Lucene
- LINQ to Metaweb(freebase)
- LINQ to Parallel (PLINQ)
- LINQ to RDF Files
- LINQ to Sharepoint
- LINQ to SimpleDB
- LINQ to Streams
- LINQ to WebQueries
- LINQ to WMI
    - http://tomasp.net/blog/linq-expand.aspx
    - http://tomasp.net/blog/linq-expand-update.aspx
- LINQ to XtraGrid

Conclusion

LINQ est une technologie fantastique dont je ne cesse de dire du bien (et d'utiliser avec bonheur dans la totalité des applications que j'écris), je ne peux imaginer utiliser demain ou dans 10 ans un langage qui n'implémentera pas une feature de même type. LINQ to "n'importe quoi" est la preuve que cette technologie est en plus ouverte et que seule notre imagination est la limite. Borland utilisait le slogan "the sky is the limit" durant ses grandes années de gloire. Microsoft ne le dit pas, mais la firme de Redmond rend ce rêve possible aujourd'hui...

Mettre des données en forme en une requête LINQ

Losqu'on traite de LINQ, la majorité des exemples utilisent des sources de données "bien formées" : flux RSS (mon dernier billet), collections d'objets, etc. Mais dans la "vraie vie" les sources de données à traiter sont parfois moins "lisses", moins bien formées. LINQ peut aussi apporter son aide dans de tels cas pour transformer des données brutes en collection d'objets ayant un sens. C'est ce que nous allons voir dans ce billet.

Prenons un exemple simple : imaginons que nous récupérons une liste de rendez-vous depuis une autre application, cette liste est constituée de champs qui se suivent, les initiales de la personne, son nom et l'heure du rendez-vous. Tous les rendez-vous se suivent en une longue liste de ces trois champs mais sans aucune notion de groupe ou d'objet.

Pour simplifier l'exemple, fixons une telle liste de rendez-vous dans un tableau de chaînes :

```
string[] data = new[] { "OD", "Dahan", "18:00", "MF", "Furuta", "12:00", "BG",
"Gates", "10:00" };
```

La question est alors comment extraire de cette liste "linéaire" les trois objets représentant les rendez-vous ?

La réponse passe par trois astuces :

- L'utilisation de la méthode Select de LINQ qui sait retourner l'index de l'entité traitée
- La syntaxe très souple de LINQ permettant d'utiliser des expressions LINQ dans les valeurs retournées
- Et bien entendu la possibilité de créer des types anonymes

Ce qui donne la requête suivante :

```
var personnes = data.Select ( (s, i) => new
                                        {
                                           Value = s,
                                           Bloc = i / 3
                                        }
```

```
                                        ).GroupBy(x => x.Bloc)
                                        .Select ( g => new
                                                    {
                                                      Initiales = g.First().Value,
                                                      Nom =
                                                          g.Skip(1).First().Value,
                                                      RendezVous =
                                                          g.Skip(2).First().Value
                                                     } );
```

L'énumération suivante :

```
foreach (var p in personnes.OrderBy(p=>p.Nom)) Console.WriteLine(p);
```

donnera alors cette sortie console de toutes les personnes ayant un rendez-vous, classées par ordre alphabétique de leur nom :

```
{ Initiales = OD, Nom = Dahan, RendezVous = 18:00 }
{ Initiales = MF, Nom = Furuta, RendezVous = 12:00 }
{ Initiales = BG, Nom = Gates, RendezVous = 10:00 }
```

Voici des objets bien formés (on pourrait ajouter un DateTime.Parse à la création du champ RendezVous pour récupérer une heure plutôt qu'une chaîne) qui pourront être utilisés pour des traitements, des affichages, une exportation, etc...

LINQ to Object ajoute une telle puissance à C# que savoir s'en servir au quotidien pour résoudre tous les petits problèmes de développement qui se posent permet réellement d'augmenter la productivité, ce que tous les autres langages et IDE promettent fallacieusement. Ici, essayez d'écrire le même code sans LINQ, vous verrez tout suite que le gain de productivité et de fiabilité est bien réel, et que la maintenance aura forcément un coup moindre.

Traiter un flux RSS en 2 lignes ou "les trésors cachés de .NET 3.5"

.NET 3.5 apporte beaucoup de classes nouvelles et d'améliorations à l'existant. Certains ajouts sont plus médiatisés que d'autres. Mais il serait injuste de limiter cette mouture à LINQ ou aux arbres d'expressions, aussi géniales et puissantes soient ces avancées.

.NET 3.5 apporte réellement une foule de nouveautés parmi lesquelles il faut noter :

- L'apport de WCF et de LINQ au compact framework
- De nouvelles facilités pour contrôler le Garbarge Collector comme le LatencyMode de la classe GCSettings
- L'ajout de l'assemblage System.NetPeerToPeer.Collaboration qui permet d'utiliser les infrastructures de peer-to-peer
- Des améliorations importantes de WCF, l'intégration WCF-WF
- etc...

Pour une liste complète des nouveautés il est intéressant de jeter un oeil à cette page MSDN.

Un exemple qui illustre les avancées plus ou moins méconnues de .NET 3.5, l'espace de noms `System.ServiceModel.Syndication` dans la dll `System.ServiceModel.Web` apporte de nouvelles facilités pour gérer des **flux RSS**. Et pour s'en convaincre quelques lignes de codes :

```
SyndicationFeed feed;
using (var r = XmlReader.Create(http://www.e-naxos.com/Blog/syndication.axd))
{ feed = SyndicationFeed.Load(r); }
```

C'est tout ! Dans la variable "feed" on a tout ce qu'il faut pour travailler sur le flux RSS.

Vous pensez que je triche et que "travailler sur le flux RSS" c'est certainement pas si simple que ça ? Bon, allez, c'est parce que c'est vous : compliquons même la chose et, à partir de ce flux, affichons le titre de tous les billets qui parlent de LINQ dans leur corps. Voici le code complet près à compiler :

```
using System;
using System.Linq;
using System.ServiceModel.Syndication;
using System.Xml;

namespace NET35RSS
{ class Program
  {
     static void Main()
     {
        SyndicationFeed feed;
        using (var r = XmlReader.Create(
                        http://www.e-naxos.com/Blog/syndication.axd))
        { feed = SyndicationFeed.Load(r); }
```

```
            if (feed==null) { Console.WriteLine("Flux vide."); return; }
            Console.WriteLine(feed.Title.Text);
            Console.WriteLine(feed.Description.Text+"\n");
            var q = from item in feed.Items
            where item.Summary.Text.ToUpper().Contains("LINQ") select item;
            foreach (var item in q) Console.WriteLine(item.Title.Text);
        }
    }
}
```

Ajoutez une petite saisie pour le mot à chercher au lieu d'un codage en dur ("LINQ" dans cet exemple) et un petit fichier paramètre pour y stocker la liste des blogs que vous connaissez, et en deux minutes vous aurez un puissant outil de recherche capable de vous lister tous les billets de vos blogs préférés qui parlent de tel ou tel sujet...

LINQ et Réflexion pour obtenir les valeurs de System.Drawing.Color

LINQ... l'un de mes sujets préférés... Plus je l'utilise, à toutes les sauces, et plus je trouve cette technologie indispensable. Mais foin de propagande, voici encore un exemple qui prouve à quel point LINQ peut servir à tout, facilement.

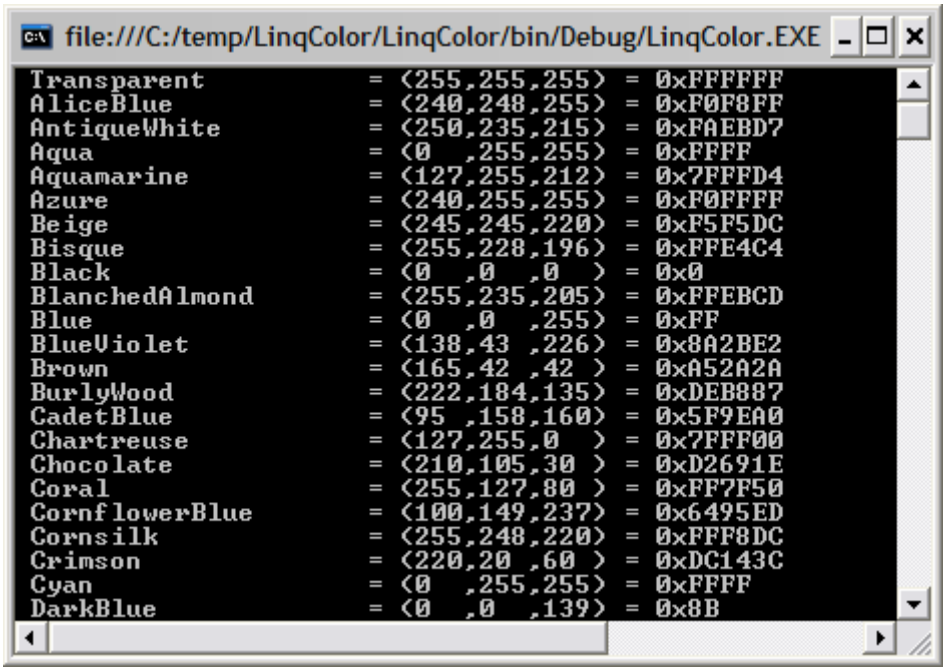
Le but du jeu : récupérer la définition des couleurs de `System.Drawing.Color`, en faire un tableau trié par ordre alphabétique.

A la "main" je vous souhaite bien du plaisir... Avec LINQ cela donne la chose suivante :

```
namespace LinqColor
{
  class Program
  {
     static void Main(string[] args)
    {
       var color_type = typeof(System.Drawing.Color);
       var color_properties = from prop in color_type.GetProperties()
                              where prop.PropertyType == color_type
                              orderby prop.Name
                              select prop;
       foreach (var color_property in color_properties)
       {
           var cur_color = (System.Drawing.Color)
                           color_property.GetValue(null, null);
```

```
            uint rgb_int = (uint) cur_color.ToArgb() & (0x00ffffff);
            Console.WriteLine(" {0,-20} = ({1,-3},{2,-3},{3,-3}) =
0x{4:X00000000}",
            color_property.Name,
            cur_color.R, cur_color.G, cur_color.B,rgb_int);
        }
    }
  }
}
```

La sortie écran (console) donne la chose suivante :



Le projet Visual Studio pour les fainéants du clavier : LinqColor.zip (5,43 kb)

Un cadeau à télécharger : des exemples LINQ !

Dernièrement je vous parlais des "101 exemples LINQ" publiés par Microsoft et je vous donnais les liens vers la page Web de ces derniers ainsi que vers un téléchargement d'une application complète distribuée avec la CTP de mars 2007. Ces exemples sont toujours d'une grande pertinence, LINQ étant toujours tel qu'il a été créé. Son importance capitale dans tout code moderne renforce même celle de ces exemples souvent méconnus alors que si utiles. Ces 101 exemples représentent aussi un excellent aide-mémoire de la syntaxe LINQ.

Hélas, qu'il s'agisse du site Web ou bien de l'appli en question, le code n'est pas tout à fait fonctionnel. On peut toujours le lire pour regarder les exemples de syntaxe

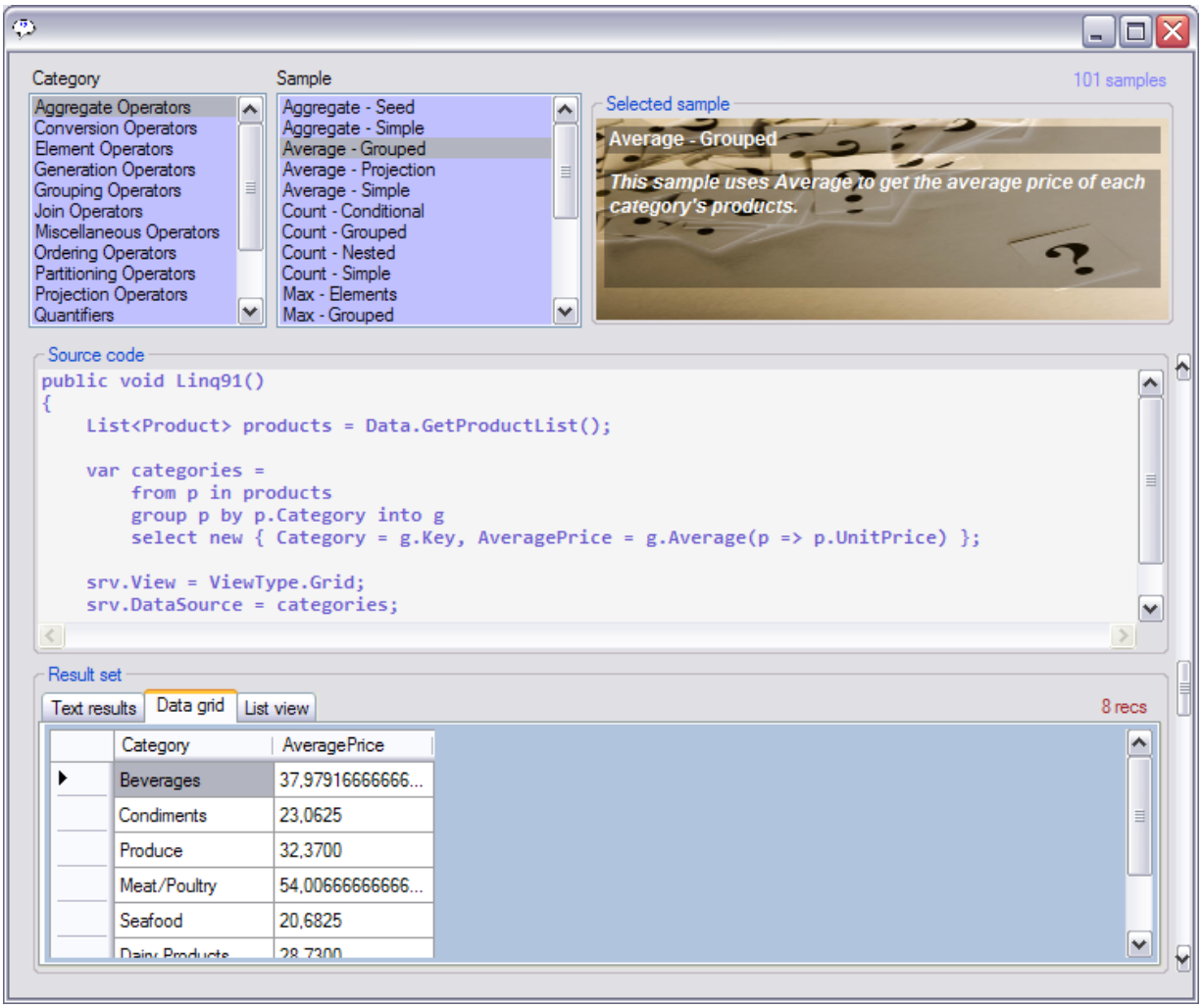
LINQ mais il est quand même plus agréable de disposer d'une application qui marche pour tester.

A l'époque de la sortie des "101 exemples" j'avais commencé à me bricoler une application. En partant des exemples publiés sur le site Web j'avais construit une appli de test "laboratoire" pour expérimenter des tas de choses pas forcément liées à LINQ. Quand j'ai étudié plus tard l'exemple MS de la CTP de mars 2007 je me suis aperçu que face aux mêmes problèmes le concepteur de cette dernière et moi avions finalement trouvé des solutions très proches. Et comme la version MS gérait deux ou trois choses que je n'avais pas implémentées j'ai décidé, en raison des ressemblances des codes, de les fusionner en une application simple et fonctionnelle.

Ce week-end le temps n'étant pas vraiment à la baignade (l'eau est bonne mais le vent de nord... qui interdit d'aller à la pêche aussi...) j'ai ressorti ce projet que j'avais laissé dans un coin de disque. Je l'ai dépoussiéré et finalisé pour être utilisable. Au passage cela m'a permis de vérifier la bonne tenue de route de Resharper qui gère vraiment bien les nouveautés de C# 3.0 et de LINQ (je vous en parlais aussi dernièrement).

Le code est téléchargeable ici (code source + l'exécutable dans bin/debug) : Linq101.rar (488,53 kb)

Une petite image de l'interface pour la route :



Bon téléchargement et bons tests !

Linq au quotidien, petites astuces et utilisations détournées...

Linq je vous en parle souvent, c'est tellement génial. Et ce que j'aime bien c'est trouver des utilisations presque à contre-sens de Linq.
J'entends par là des façons de "rentabiliser" Linq à toute autre tâche que d'interroger des données (SQL, XML ou autres).

Je ne suis pas le seul dans ce cas ! J'ai au moins trouvé un autre zigoto qui s'amuse au même jeu, Igor Ostrovsky dont vous pouvez visiter le blog (cliquez sur son nom). Comme il s'exprime en anglais et que je sais que certains visiteurs ici ne sont pas forcément doués pour cette langue, voici une adaptation libre(*) de son billet présentant quelques astuces très amusantes.

(*) je n'ai repris que les exemples de code, le texte et le projet exemple sont de mon cru, si vous parlez anglais je vous laisse découvrir la prose de Igor sur son blog.

Initialiser un tableau, une liste

Il est souvent nécessaire d'initialiser une array avec des éléments qui, pour que cela soit utile, répondent à certains critères. Normalement cela se règle soit sous la forme d'une liste de constantes dans la déclaration de l'array, ce qui est pour le moins assez peu souple, soit sous la forme d'un code d'initialisation, ce qui du coup devient assez lourd pour parfois pas grand-chose... C'est oublier que Linq peut apporter ici une aide non négligeable !

Voici trois exemples :

```
int[] A = Enumerable.Repeat(-1, 10).ToArray();

int[] B = Enumerable.Range(0, 10).ToArray();

int[] C = Enumerable.Range(0, 10).Select(i => 100 + 10 * i).ToArray();
```

Les trois arrays contiendront 10 éléments. "A" contiendra 10 éléments tous égaux à -1. "B" contiendra la suite 0 à 9, et "C" contiendra la suite de 100 à 190 par pas de 10.

Amusant, n'est-il pas ?

Plusieurs itérations en une seule

Imaginons que nous possédions deux arrays, A1 et B2, chacune contenant une sélection différente d'éléments. Supposons que souhaitions appliquer le traitement "Calcule(x)" aux éléments de ces deux arrays. La solution s'imposant serait :

```
foreach (var x in A1) { Calcule(x); }
foreach (var x in A2) { Calcule(x); }
```

Ce n'est pas très long, mais répéter du code n'est jamais très élégant. Voici comme Linq permet de faire la même chose en une seule boucle :

```
foreach (var x in A1.Concat(A2)) { Calcule(x); }
```

L'opérateur `Concat` effectue une .. concaténation, donnant l'illusion à notre boucle qu'elle travaille sur une seule et même array alors qu'il n'en est rien... On notera que Linq travaillant au niveau des énumérateurs, il n'y même pas de variable temporaire créée par le mécanisme pour stocker la concaténation. C'est élégant et en plus c'est efficace du point de vue de la mémoire.

Fun, isn't it ?

Créer des séquences aléatoires

Il est parfois nécessaire de créer des séquences aléatoires, pour tester un bout de code par exemple. La classe Random est bien pratique mais encore une fois il faudra déclarer le tableau puis déclarer une méthode pour l'initialiser en utilisant Random. Avec Linq tout cela peut se réduire à deux déclarations :

```
Random rand = new Random();
var ListeAléatoire = Enumerable.Repeat(0, 10).Select(x => rand.Next(10,500));
```

Le tableau "`ListeAléatoire`" contiendra 10 nombres aléatoires entre 10 et 500, et pas une seule déclaration de méthode à l'horizon. Vous noterez en revanche l'utilisation qui est faite de l'expression Lambda (nouveauté de C# 3.0) au niveau du Select.

Créer des chaînes de caractères

Si les exemples portent jusqu'ici sur des tableaux de nombres, c'est par pure souci de simplification. Il est bien entendu possible d'appliquer les mêmes astuces à d'autres types de données comme les chaînes :

```
 string chaine = new string( Enumerable.Range(0, 20) .Select(i => (char)('A' + i % 3)) .ToArray());
```

La chaîne "chaine" contiendra "ABCABCABCABCABCABCAB". Remplacez le "20" par un "rand.Next(10,100)" par exemple, et en plus la longueur de la chaîne sera elle-même aléatoire. Ce n'est pas pratique ça ?

Dans les commentaires du blog d'Igor, un lecteur proposait une autre façon d'initialiser une chaîne avec Linq, la voici pour l'intérêt de la démonstration de la versatilité de Linq :

```
string chaine2 = string.Join(string.Empty, Enumerable.Repeat("ABC", 7).ToArray());
```

Ici on peut passer un pattern et un nombre de répétition.

Convertir une valeur en séquence de 1 objet

Il est parfois nécessaire de convertir une valeur en une séquence de longueur 1. Passer un paramètre unique là où un tableau de valeur est attendu réclame de créer le fameux tableau et de l'initialiser. Rien de bien méchant mais pourquoi ne pas mettre Linq à contribution ?

```
IEnumerable<int> sequence = Enumerable.Repeat(LaValeur, 1);
```

La répétition de 1 élément peut sembler être une sorte de contresens, mais si les concepteurs de la fonction partageaient cette opinion n'auraient-ils pas levé une exception pour toute valeur inférieure à 2 ... :-)
C'est en tout cas un moyen bien pratique d'arriver au résultat souhaité en une seule opération !

Enumérer tous les sous-ensembles d'un tableau

Il s'agit plus ici d'un "plaisir pervers" que d'une réelle utilité quotidienne il faut l'avouer, mais pour l'intérêt de la syntaxe voici comment énumérer tous les sous-ensembles d'un tableau (attention, si la taille du tableau dépasse quelques dizaines d'éléments le temps de calcul peut vite exploser ...) :

```
T[] tableau = …;
var subsets = from m in Enumerable.Range(0, 1 << tableau.Length)
              select
                  from i in Enumerable.Range(0, tableau.Length)
                  where (m & (1 << i)) != 0
                  select arr[i];
```

Je vous laisse réfléchir à la syntaxe, peu évidente de prime abord, et le côté un peu "tordu" de cette astuce, mais elle répond à un besoin qui l'est tout autant. Et Linq peut aussi servir à cela.
Quelques clés pour comprendre : la requête ne retourne pas des items de "tableau" mais bien des sous-ensembles de "tableau", c'est à dire des tableaux. Chaque tableau contient une combinaison unique d'éléments de "tableau". Si vous prenez tableau = B de notre premier exemple (c'est à dire la suite d'entiers 0 à 9), la requête Linq ci-dessus retournera 1024 sous-ensembles. Avouez que bien qu'un peu difficile à

comprendre à première lecture, le procédé est d'une grande élégance... (Vous pouvez regarder le fonctionnement de cette requête de plus près grâce au projet démo téléchargeable en fin d'article).

Un peu de code...

Pour mieux comprendre les choses il est toujours préférable de faire tourner un bout de code. J'y ai pensé ! Vous trouverez un projet console complet plus bas ;-)

Conclusion

Igor montre au travers de ces différentes utilisations de Linq ce que je tente aussi de vous montrer dans mes billets : Linq est d'une grande souplesse, Linq ne se limite pas à faire du SQL en C#, Linq est un atout majeur pour le développeur qui sait s'en servir au quotidien. Initialiser un tableau, convertir un tableau dans un type différent, jouer sur les combinatoires, fabriquer des chaînes de caractères ou des séquences aléatoires de test, etc, etc. Il n'y a pas un bout de code dans lequel il ne soit possible de glisser un peu de Linq pour le rendre plus sobre, plus élégant et souvent plus performant.
Remercions Igor pour ces exemples de code (seule chose reprise de son billet, j'ai mon propre style, il a le sien :-))

Le code complet
LinqFun.rar (99,18 kb)

Bonus
Mitsu, qu'on ne présente plus, ça serait faire insulte à sa notoriété !, a proposé il y a quelques temps déjà une autre astuce Linq tout à fait amusante, elle consiste à obtenir la décomposition d'un nombre en facteurs premiers au travers de requêtes Linq et d'un class helper permettant d'écrire : 144.ToPrimeNumbersProduct() et d'obtenir en sortie "2^3 * 3^2 * 7". Je vous en conseille la lecture en suivant ce lien.

Donner du peps à Linq ! (Linq dynamique et parser d'expressions)

J'ai eu moult fois ici l'occasion de dire tout le bien que je pense de LINQ, de sa puissance, de sa souplesse, de sa versatilité (de XML à SQL en passant par les objets et l'Entity Framework).

Mais une petite chose me chagrinait : toute cette belle puissance s'entendait "*hard coded*". Je veux dire par là que les expressions et requêtes LINQ s'écrivent en C# dans le code et qu'il semblait très difficile de rendre la chose très dynamique, sous-entendu dépendant d'expressions tapées à l'exécution de l'application. **En un mot, faire du LINQ dynamique comme on fait du SQL dynamique**.

Ce besoin est bien réel et ne concerne pas que les sources SQL. Prenez une simple liste d'objets que l'utilisateur peut vouloir trier selon l'une des propriétés ou filtrer selon le contenu des objets... *Comment implémenter une telle feature ?*

Contourner ce manque en écrivant des parsers, en jonglant avec les expressions lamba et la réflexion n'est pas, il faut l'avouer, ce qu'il y a de plus simple. Mais rassurez-vous chez Microsoft des gens y ont pensé pour nous ! Seulement voilà, la chose est assez confidentielle il faut bien le dire, et on ne tombe pas dessus par hasard. Ou alors c'était un jour à jouer au Lotto, voire à contrôler, selon les mauvaises langues, l'emploi du temps de sa femme !

La chose s'appelle "**LINQ Dynamic Query Library**", un grand nom pour une petite chose puisqu'il s'agit en réalité d'un simple fichier source C# à ajouter à ses applications. Mais quel fichier source !

Toute la difficulté consiste donc à savoir que ce fichier existe et mieux, où il se cache... C'est là que c'est un peu vicieux, la chose se cache dans un sous-répertoire d'un zip d'exemples Linq/C#, fichier lui-même astucieusement planqué dans la jungle des milliers de téléchargement de MSDN...

Sans plus attendre allez chercher le fichier cliquant ici.

Une fois que vous aurez accepté les termes de la licence ("I Accept" tout en bas à gauche) vous recevrez le fichier "CSharpSamples.zip". Dans ce dernier (dont tout le contenu est vraiment intéressant) aller dans le répertoire "LinqSamples" puis dans le projet "DynamicQuery" vous descendrez dans une autre sous-répertoire appelé lui aussi "DynamicQuery" (non, ce n'est pas un jeu de rôle, juste un téléchargement MSDN!). Et là, le Graal s'y trouve, "Dynamic.cs".

Copiez-le dans vos projets, et ajouter un "using System.Linq.Dynamic". A partir de là vous pourrez utiliser la nouvelle syntaxe permettant de faire du LINQ dynamique. A noter que dans le répertoire de la solution vous trouverez aussi un fichier "Dynamic Expressions.html" qui explique la syntaxe du parser.

Quelles sont les possibilités de LINQ Dynamic Query Library ?

C'est assez simple, pour vous expliquer en deux images je reprend deux captures écrans tirées du blog deScott Guthrie (un excellent blog à visiter si vous parlez l'anglais).

Voici un exemple de requête LINQ (en VB.NET, si j'avais refait la capture cela aurait été du C#, mais je ne vais pas rouspéter hein !)

```
Dim Northwind As New NorthwindDataContext

Dim query = From p In Northwind.Products _
            Where p.CategoryID = 2 And p.UnitPrice > 3 _
            Order By p.SupplierID _
            Select p

GridView1.DataSource = query
GridView1.DataBind()
```

Maintenant voici la même requête utilisant du LINQ dynamique :

```
Dim Northwind As New NorthwindDataContext

Dim query = Northwind.Products _
                     .Where("CategoryID=2 And UnitPrice>3") _
                     .OrderBy("SupplierId")

GridView1.DataSource = query
GridView1.DataBind()
```

Vous voyez l'astuce ? Le filtrage et le tri sont maintenant passés en chaîne de caractères... Bien entendu cette chaîne peut avoir été construite dynamiquement par code (c'est tout l'intérêt) ou bien saisie depuis un TextBox rempli par l'utilisateur.

Ici pas de risque d'attaque par "SQL injection", d'abord parce que LINQ n'est pas limité à SQL et que le problème ne se pose que dans ce cas, mais surtout parce que LINQ to SQL utilise des classes issues d'un modèle de données "type safe" et que par défaut LINQ to SQL est protégé contre les attaques de ce type. Pas de souci à se faire donc.

Pour terminer j'insisterai lourdement sur le fait que LINQ ne se limite pas aux données SQL, et que l'astuce de LINQ dynamique ici décrite s'applique même à LINQ

to Object par exemple. Laisser la possibilité à un utilisateur de trier une List<> ou une Collection<>, de la filtrer, tout cela en mémoire et sans qu'il y ait la moindre base de données est bien entendu possible.

LINQ est vraiment une innovation majeure en matière de programmation, pouvoir le dynamiser simplement c'est atteindre le nirvana...

Je vous laisse vous jeter sur vos claviers pour expérimenter du LINQ dynamique, amusez-vous bien !

LINQ à toutes les sauces !

J'ai eu maintes fois l'occasion de vous dire ici que LINQ est l'innovation la plus fantastique que j'ai vue depuis longtemps dans un langage. Pour vous prouver que LINQ peut servir partout et tout le temps voici deux exemples à contre-courant de l'idée qu'on se fait des utilisations possibles de LINQ :

Cas 1 : Lister les services actifs de Windows.

C'est bête mais balayer et filtrer une simple liste comme celle-là (et de bien d'autres du même genre retournées par le Framework .NET ou par vos applications), c'est produire du code pas très marrant... Avec LINQ ça devient :

```
using System.ServiceProcess;
var srv = from s in ServiceController.GetServices()
          where s.Status == ServiceControllerStatus.Running
          select s.DisplayName;
ListBox1.DataSource = srv.ToList();
```

Je trouve ça élégant, pas vous ?

Cas 2 : Remettre à unchecked tous les Checkbox d'une form

Balayer certains contrôles d'une fiche n'est là non plus pas l'endroit où l'on s'attend à trouver du LINQ... Et pourtant ! Imaginons une fiche de saisie avec des tas de checkbox et un bouton "raz" de remise à zéro de la fiche. Balayer tous les contrôles de la fiche pour ne sélectionner que les checkbox n'est pas un code bien complexe mais avec LINQ ça devient tellement plus chouette !

```
var cb = (from Control c in this.Controls select c).OfType<CheckBox>();
foreach (var c in cb) c.Checked=false;
```

Ce n'est pas plus joli et plus clair écrit comme ça ? (et encore on pourrait se passer de la variable "cb" et intégrer directement la requête LINQ après le "in" du "foreach")?

LINQ Join sur plusieurs champs

Linq et ses merveilles... Linq et ses jointures... Parlons-en des jointures : savez-vous comment faire une jointure sur plusieurs champs à la fois ?

Join on / Equals

Pour faire une jointure avec Linq, c'est facile, il suffit d'utiliser JOIN ON, un peu comme en SQL.

Prenons un petit exemple simple, l'exemple 103 des fameux "101 exemples" (il y en a un petit plus que 101 mais l'histoire de ces exemples a été débattu il y a longtemps – voir mon billet 101 exemples ou vous trouverez l'application exemple que j'avais écrite et qui est bien utile pour se rappeler des différentes syntaxes !). Le voici l'exemple 103 :

```
public void Linq103()

{

    string[] categories = new string[]{

        "Beverages",

        "Condiments",

        "Vegetables",

        "Dairy Products",

        "Seafood" };


    List<Product> products = Data.GetProductList();


    var q =

        from c in categories
```

```
        join p in products on c equals p.Category into ps
        select new { Category = c, Products = ps };


    foreach (var v in q)
    {
        Txt+=string.Format(v.Category + ":");
        foreach (var p in v.Products)
        {
            Txt+=string.Format("    " + p.ProductName);
        }
    }
}
```

Ce qui est intéressant c'est bien entendu la requête (var q=...).

Comme on le constate le JOIN s'écrit un peu comme le FROM avec une clause ON en plus. Cette dernière indique la paire de champs à considérer pour créer la jointure.

On notera ici le stockage intermédiaire dans "ps" ce qui n'est qu'une possibilité, pas une obligation (regardez mon application exemple des 101 exemples et vous trouverez les nombreuses variantes de JOIN, et du reste de la syntaxe Linq).

Première chose : il faut utiliser "equals" et non le signe "==".

Seconde chose : c'est une paire de champs, un de chaque entity set (en gros un de chaque "table").

Mais avec deux champs ?

C'est facile se dit-on, avec deux champs je rajoute "&&" suivi de la seconde paire. Quelque chose du style :

```
join p in products on c.Field1 equals p.FieldA && c.Field2 equals p.FieldB
```

Perdu !

Hélas, ça ne passe pas. Le JOIN ne marche que sur UNE paire de champs.

L'astuce

Il y a en a une, vous le savez déjà, sinon je n'écrirais pas ce billet... Et elle passe par la création d'une classe anonyme, du moins d'instances de classe anonyme.

L'idée est de conserver une paire de valeurs qui peuvent être comparées par un Equals. Toute la ruse se situe dans le fait qu'au lieu d'un champ simple nous allons créer une instance anonyme comportant l'ensemble des champs à tester. Ainsi l'exemple (faux) donné juste ci-dessus s'écrira en réalité :

```
join p in products on

   new { c.Field1, c.Field2 }

    equals

   new { p.FieldA, p.FieldB }
```

Futé non ?

Un petit détail qui n'est pas évident ici : comme on créée une instance de classe anonyme, on peut, comme ci-dessus, mettre directement les champs les uns derrière les autres avec une virgule. Cela va créer automatiquement un type anonyme dont les propriétés porteront les mêmes noms.

Or, dans l'exemple que je viens de donner, on trouve des "c.Field1" et "c.Field2" d'un côté et de l'autre des "p.FieldA" et "p.FieldB". Cela ne marchera pas... Car les deux instances anonymes ne seront pas compatibles entre elles puisque la première contiendra deux propriétés nommées "Field1" et "Field2" alors que le second type aura des propriétés s'appelant "FieldA" et "FieldB".

Très souvent les JOIN s'expriment sur des champs (plus exactement pour Linq, des Propriétés) qui portent le même nom. En tout cas cela devrait être le cas... Mais la réalité étant un peu différente de l'idéal, il arrive tout aussi souvent que les noms de propriétés dans les deux classes à comparer ne soient pas rigoureusement identiques.

C'est foutu alors ?

Mais non. Il suffit d'adopter une syntaxe plus verbeuse pour l'instance anonyme en indiquant clairement le nom de ses propriétés. Ainsi l'exemple devient syntaxiquement correct en l'écrivant ainsi :

```
join p in products on

    new { ChampA=c.Field1, ChampB=c.Field2 }

    equals

    new { ChampA=p.FieldA, ChampB=p.FieldB }
```

Et voilà... En indiquant explicitement les noms des propriétés des instances anonymes on s'assure que le compilateur traitera bien les deux "new" comme faisant appel à la même classe de base, anonyme certes, mais la même.

Linq to Entities : ".Date" non supporté – Une solution

Quand on utilise Wcf Ria Services on utilise souvent côté serveur Linq To Entities. Et quand on utilise Linq on aime bien tester ses requêtes avec l'outil LinqPad. Voici un ménage à trois qui nous réserve parfois des surprises, des bonnes et des moins bonnes...

Séparons bien les problèmes

Le problème principal est que Linq To Entities ne supporte pas le ".Date" des DateTime.

Le fait d'utiliser Linq To Entities avec les Wcf Ria Services (avec Silverlight) n'est qu'un cas d'utilisation parmi d'autres. On peut utiliser L2E même dans une application Console ou dans une application WCF. Il s'agit juste d'une complication qui rend le debug plus délicat.

Quant à LinqPad, c'est un outil dont je vous ai certainement déjà parlé tellement il est pratique. Et si vous ne connaissez pas, suivez le lien et téléchargez (c'est gratuit) et vous me remercierez plus tard...

Mais voilà, LinqPad utilise Linq To SQL pour se connecter aux données. Linq To SQL est en quelque sorte l'ancêtre de Linq To Entities et on s'attend à voir passer dans ce dernier ce qui passe aussi dans le premier. Or, il faut comprendre que Linq To SQL a été écrit en adéquation avec SQL Server, alors que Linq To Entities est conçu pour supporter différents serveurs (ce qui en fait, en partie, l'avantage). [*Nota : les dernières versions de LinqPad savent utiliser vos propres DLL pour interroger des données via Linq to Entities*]

Quand on travaille sur une application utilisant Linq to Entities, on ne va pas se priver d'utiliser LinqPad pour tester ses requêtes, cela serait idiot. Mais parfois on oublie que ce n'est pas du Linq To Entities. Dès lors, certaines requêtes bien parfumées et épilées de près qui tournent comme une ballerine sur le Casse-Noisette se plantent lamentablement tel un volatile malchanceux jouant le final du Lac des Cygnes. Et tout cela uniquement devant la salle pleine et non pendant les répétitions – je veux dire au runtime et non à la compilation.

Un avertissement pour commencer

Ecrire cent fois :

"Je ne prendrais pas pour argent comptant les tests de requêtes effectués sous LinqPad lorsque je travaille sous Linq To Entities".

C'est un moyen un peu vieillot et totalement idiot mais redoutablement efficace pour retenir une leçon...

Revenons au problème

Bref, il y a un problème, L2E ne supporte pas ".Date" alors que Linq to SQL le supporte.

Vous allez me dire, on s'en fiche un peu non ?

Non, pas du tout car dans des applis de gestion (des LOB's disent en ce moment les américains - et les français qui aiment faire les malins à la machine à café) il n'est pas rare du tout de faire des choses comme un groupage de données sur la date...

Et un groupage sur la date veut dire sur la date, non pas sur la date et l'heure (type DateTime). Les groupes ne veulent plus rien dire lorsqu'on prend en compte la partie heure au millième de seconde !

Il faudrait pouvoir extraire la partie Date uniquement, et c'est justement ce que fait ".Date" de DateTime.

Du coup, une requête comme celle-ci ne passe pas sous L2E :

```
var q = from x in MaBase.MaTable

        group x by x.DateDeFacturation.Date into g

        select g;
```

Dommage...

Certains s'en sortiront très vite en ajoutant une colonne calculée à la base de données pour isoler la partie Date. Ce n'est pas forcément une mauvaise solution mais elle oblige a modifier le schéma de la base ce qui, dans de nombreuses situations, ne se fait pas comme ça tout seul dans son coin et uniquement pour simplifier l'écriture d'une requête.

La solution

Heureusement on peut aussi être filou et réfléchir un peu. Et parfois on trouve une solution qui à défaut d'être d'une grande élégance, reste simple et facile à implémenter.

Dans notre cas l'astuce consiste à créer une variable locale dans la requête et à grouper sur un type anonyme (solution souvent intéressante dans d'autres cas, comme le groupage sur plusieurs champs, normalement impossible).

La requête précédente devient alors :

```
var q = from x in MaBase.MaTable

        Let d = x.DateDeFacturation // la variable locale

        group x by new {y = d.Year, m = d.Month, d = d.Day} into g // le type
anonyme
```

```
        select g;
```

Chat échaudé…

… craint l'eau froide. C'est bête mais je me rappelle que quand j'étais petit je ne comprenais pas ce proverbe. Je ne voyais pas en quoi un pauvre chat qui a eu trop chaud pouvait craindre l'eau froide alors même que, au contraire, d'avoir eu trop chaud devait l'inciter à chercher la fraicheur bienfaisante d'une eau froide… J'étais trop logique, et je cherchais trop la logique partout, déjà petit j'étais informaticien… 

Passons l'anecdote et rappelons qu'en bon chat échaudé par le ".Date" nous allons nous méfier de l'eau froide qui coule dans toutes les notions de temps. Et nous avons raison ! Car de nombreux soucis existent aussi avec les TimeStamp sous L2E. Mais je vous laisse la joie de le découvrir… (Je vous aurais tout de même prévenu !).

On en revient au fait de bien tester ses requêtes à l'exécution, et donc à la tentation d'utiliser LinqPad, et à l'avertissement donné plus haut … Tout se tient 

Conclusion

Ici nous nous en sortons plutôt bien, la pirouette est possible. Dans d'autres cas L2E est parfois moins souple et il faut repenser les requêtes autrement, voir travailler en plusieurs passes avec du Linq To Object coincé comme une tranche de jambon entre deux tranches de Linq To Entities. Drôle de sandwich, surtout que souvent, il n'y manque rien puisque nous y jouons le rôle du cornichon…

Il faut rappeler une nouvelle fois que depuis ce billet LinqPad a bien évolué et autorise les interrogations via L2E. Cela simplifie grandement les choses et supprime certains petits problèmes. Mais il faut toujours resté vigilant quand on teste du code en dehors de son contexte réel car cela entraîne des petites (ou grosses) distorsions qui peuvent amener des soucis.

Créer un arbre des répertoires avec XML (et l'interroger avec LINQ to XML)

Pour les besoins d'un prochain billet je voulais obtenir une structure arborescente de test sous la forme d'un fichier XML. Une telle chose peut se faire à la main mais cela est fastidieux et à force de copier/coller les données seront trop semblables et donc

peu utilisables pour des tests et du debug, ce qui est dommage puisque tel était le but recherché...

Pour générer des données aléatoires mais réalistes je possède déjà un outil fantastique (puisque c'est moi qui l'ai fait :-)), DataGen, un logiciel puissant mais qui n'est pas étudié pour générer des données dont la structure est récursive. Restait donc à concevoir un utilitaire pour satisfaire mon besoin de l'instant.

Je n'aime pas développer "pour rien" donc il fallait que cet utilitaire en soit bien un. Tout disque dur de développeur contient une telle quantité de données que l'ensemble des répertoires forment une superbe structure arborescente ni trop petite ni trop gigantesque. Exactement ce qu'il faut pour des tests ! Ainsi, l'outil devra fabriquer un fichier XML à partir de n'importe quel niveau de répertoire, la structure étant hiérarchique.

Je m'empresse donc de vous faire partager ce petit bout de code qui vous sera utile un jour ou l'autre je pense.

```
using System;
using System.IO;
using System.Linq;
using System.Xml.Linq;


namespace Enaxos.Tools.Xml
{
    /// <summary>
    /// This class can generate an XML file from a folder hierarchy.
    /// </summary>
    public static class DirToXml
    {
        /// <summary>
        /// Builds the tree document.
        /// </summary>
```

```
/// <param name="dirname">The name of the starting folder.</param>
/// <returns>a LINQ to XML <c>XDocument</c></returns>
public static XDocument BuildTreeDocument(string dirname)
{
    return new XDocument(new XDeclaration("1.0", "utf-8", "yes"),
        new XComment("Structure au "+DateTime.Now),
                    new XElement("directories", BuildTree(dirname)));
}

/// <summary>
/// Builds the tree (recursive method).
/// </summary>
/// <param name="dirName">Name of the starting folder.</param>
/// <returns>a LINQ to XML <c>XElement</c> being the root (or 1st item)
/// of the tree.</returns>
public static XElement BuildTree(string dirName)
{
    var di = new DirectoryInfo(dirName);
    var files = di.GetFiles();
    var dirsize = 0l;
    foreach (var file in files)
    {
        dirsize += file.Length;
    }
    var subdirs = di.GetDirectories();
    // each item is a "directory" having 5 attributes
    // name is the name of the folder
    // fullpath is the full path including the name of the folder
    // size is the size of all files in the folder (in bytes)
```

```
            // files is the number of files in the folder
            // subdirs is the count of possible sub directories in the folder
            var elem = new XElement("directory",
                new XAttribute("name", di.Name),
                new XAttribute("fullpath",dirName),
                new XAttribute("size", dirsize),
                new XAttribute("files",files.Count()),
                new XAttribute("subdirs",subdirs.Count()));
            foreach (var dinf in subdirs)
            {
                var elemDir = BuildTree(dirName + "\\" + dinf.Name);
                elem.Add(elemDir);
            }
             return elem;
        }
    }
}
```

Ce code peut être utilisé sans passer par un fichier disque puisqu'on récupère le document en mémoire. L'exemple ci-dessous montre comment appeler le code et comment interroger la structure via Linq To Xml pour connaître les répertoires vides (n'ayant ni fichier ni sous répertoire) :

```
1:  var d = DirToXml.BuildTreeDocument(@"D:\WpfToolkit");
2:  d.Save(@"d:\test.xml");
3:  Console.WriteLine(d.ToString());
4:  Console.WriteLine(new string('-',60));
5:
6:  var q = from e in d.Descendants("directory")
7:          where (int) e.Attribute("files") == 0
```

```
 8:                 && (int)e.Attribute("subdirs") == 0
 9:                 orderby (string) e.Attribute("fullpath")
10:                 select e.Attribute("fullpath");
11:
12:  Console.WriteLine("Répertoires vides");
13:  foreach (var element in q)
14:    { Console.WriteLine(element); }
15:  Console.WriteLine(string.Format("{0} répertoires vides",q.Count()));
```

ADO.Net Entity Framework - une série d'articles en US

Stefan Cruysberghs a publié (en anglais) une série de trois articles (récents) sur l'Entity Framework qui proposent un tour d'horizon assez intéressant de cette technologie. En voici le sommaire (avec accès direct aux articles):

- The Entity Framework architecture
- Installation
- Generate an Entity Data Model
- The Entity Data Model (EDM)
    - LINQ to SQL Diagram vs. Entity Data Model
    - Names of EntityTypes and EntitySets
    - Scheme file (EDMX)
    - Model Browser
    - Mapping Details
    - Generated entity classes
    - Documentation
- Entity SQL
    - ObjectQuery queries which return entity types
    - ObjectQuery queries with parameters
    - ObjectQuery queries which return primitive types
    - ObjectQuery queries which return anonymous types
    - EntityCommand queries which return entity types

    - EntityCommand queries which return anonymous types
    - EntityCommand queries with parameters
- LINQ to Entities
    - LINQ queries with parameters
    - LINQ queries which return anonymous types

Part 2 :

- View SQL statements
    - SQL profiling tools
    - ToTraceString method
- Tools
    - eSqlBlast
    - LINQPad

Part 3 :

- Add, update and delete entities
    - Update (modify) an entity
    - Add (create) and entity
    - Add (create) entity with associated entities
    - Delete an entity
- Concurrency handling
    - Set ConcurrencyMode
    - Resolve concurrency conflicts
- Change tracking
    - ObjectStateManager
    - Dump() & DumpAsHtml() extension methods
    - NoTracking option

Si vous préférez accéder aux trois articles séparément :

- .NET - ADO.NET Entity Framework & LINQ to Entities - part 1
- .NET - ADO.NET Entity Framework & LINQ to Entities - part 2
- .NET - ADO.NET Entity Framework & LINQ to Entities - part 3

Bonne lecture !

Entity Framework Application Patterns. Résumé de la conférence DAT303 Teched

Hier je vous ai résumé la conférence DAT201 qui présentait l'Entity Framework. Aujourd'hui je vous parlerai plus brièvement d'une autre conférence. Cette brièveté du résumé ne doit pas vous induire en erreur : la conférence DAT303 de Pablo Castro, Technical Lead, est peut-être **la meilleure à laquelle j'ai assisté**.

D'abord Pablo est un jeune gars sympa. Ensuite malgré un fort accent cubain ou mexicain il a soutenu une conf à un rythme d'enfer tout en étant d'une grande clarté, une vraie conf technique comme je les aime, faite par un passionné qui connaît son affaire. Enfin, cette conférence dépassait le cadre de la présentation générale pour parler vrai et pratique. Là, Pablo m'a appris des choses sur l'Entity Framework. Des trucs pas simples à découvrir tout seul, une conférence qui fait vraiment gagner du temps et de la compétence. Merci Pablo !

Si le résumé sera court c'est que 95% de la conférence de Pablo était basée sur du code. Vous imaginez bien qu'il n'était pas possible de prendre le tout en sténo dans la pénombre de la salle de conf... J'ai revisionné aujourd'hui la conf (puisque toutes les sessions principales ont été filmées), et franchement même assis dans mon bureau à l'aise il n'était pas possible de noter la totalité des manips, toutes essentielles. J'ai d'ailleurs demandé à Pablo s'il pouvait avoir la gentillesse de m'envoyer le code source de ces démos, et s'il accepte je les mettrais en téléchargement ici (je n'ai jamais eu de réponse, pas si sympa que ça le Pablo !).

Donc de quoi s'agissait-il ?

Bien entendu de l'Entity Framework, depuis mon billet d'hier vous devez savoir de quoi il s'agit (sinon foncez lire ce dernier, ne vous inquiétez je ne bouge pas, vous pourrez revenir ici plus tard ☺).

Mais il n'était plus question ici de parler en général ou même en détail du fonctionnement de EF, il s'agissait de voir comment s'en servir "*en vrai*" dans trois conditions :

- En 2 tiers
- Dans une application Web
- Au sein d'une architecture 3-tiers

En effet, le principal problème de toute surcouche est de manger un peu plus de CPU et de mémoire que la couche du dessous. C'est la règle en informatique, plus c'est pratique et puissant, plus la machine doit pédaler. Et même si EF est très efficace et très optimisé, ce qu'il fait en plus possède forcément un coût. Il est donc essentiel de pouvoir maîtriser ce coût en faisant des économies là où cela est possible.

La bonne nouvelle c'est que EF le prévoit et que la mise en œuvre est simple. La moins bonne nouvelle c'est que la chose devient un poil plus "mystique" et qu'il faut réfléchir un peu plus. Mais un informaticien paresseux des neurones est soit en chômeur en puissance, soit un ex-informaticien au RSA...

Change tracking et Identity resolution

Deux mécanismes peuvent être économisés (et les ressources CPU/RAM qui vont avec) : Le change tracking et l'identity resolution. Le premier, détection des changements, permet à l'EF de savoir quels objets ont été modifiés pour savoir comment appliquer les mises à jour à la base de données. Si certaines entités (ou grappes d'entités) ne seront pas modifiées on peut alors se passer du mécanisme de détection des changements... L'identity resolution, ou résolution des identités est utilisée pour identifier de façon formelle toutes les instances des entités. Il est en effet primordial pour le système de savoir à quel(s) enregistrement(s) de la base de données corespond(ent) l(es) entité(s) sinon il est impossible d'envisager des fonctions comme le refresh, l'update ou le delete...

Heureusement EF est très bien conçu et il est possible de stopper ces mécanismes là où on le désire. Ce court billet n'entrera pas dans les détails, je prépare un article sur la question, format plus adapté à un long exposé technique avec exemples de code.

Des entités qui passent les frontières...

Un autre problème se pose, notamment dans les applications en multi-tiers : lorsqu'une instance d'entité est passée en dehors du système, EF en perd la trace... Imaginons un serveur applicatif qui utilise EF pour accéder à une base de données afin d'offrir des services de type WCF (ex remoting, pour simplifier) ou même des services Web. Les changements dans les objets ont lieu en dehors même de la machine qui utilise EF. Le lien est cassé et si l'objet revient au serveur applicatif qui doit en retour mettre à jour la base de données, EF ne le connaîtra plus et ne pourra pas l'intégrer à son mécanisme de mise à jour de la base.

Là encore EF permet de contourner le problème. La façon de le faire sera aussi décrite dans le papier en cours de préparation, soyez patients !

Compilation LINQ

Enfin, Pablo a montré comment économiser des ressources du côté de LINQ cette fois-ci en utilisant la possibilité de compiler la requête sous la forme d'un delegate qu'il suffit ensuite d'appeler. L'utilisation des expressions Lambda et l'utilisation des interfaces idoines permettent même d'avoir une requête compilée mais paramétrique dont le résultat peut être réutilisé dans une autre requête LINQ. Pas de magie, et peu de code à taper pour réaliser tout cela. En revanche, une fois encore, cela ne s'improvise pas si on ne connaît pas l'astuce.

Conclusion

Une conférence riche, dense et instructive. Je pourrais en parler encore des pages entières mais sans la contrepartie du code exemple et des explications qui vont avec ce billet deviendrait vite ennuyeux. Je préfère m'arrêter là et réserver les détails d'implémentation pour l'article que je prépare sur ce sujet. Bien entendu il sera annoncé ici et sera téléchargeable gratuitement comme d'habitude.

Une dernière chose, Pablo nous a fait voir un peu Astoria, un procédé permettant d'exposer une (partie d'une) base de données en HTTP dans un formalisme XML. Une sorte de Web service généré automatiquement pour chaque classe et qui, par des GET ou des POST permet d'accéder aux données mais aussi de mettre à jour les données dans un mécanisme de type accès à une page Web ! Je n'ai pas eu le temps de creuser la question ni de faire tourner de bêta de Astoria, mais soyez sûrs que dès que j'en aurais fini avec mon tri de toutes les conférences des TechEd, Astoria sera au programme ! [Le projet Astoria deviendra ADO.Net Data Services puis en 2009 WCF Data Services. Il utilise le protocole OData. Un petit projet qui a donc fait son chemin depuis 2007 étant donné l'incroyable utilité de WCF Data Services !]

Entity Framework - Résumé de la session DAT201 aux TechEd

Cette conférence était l'une de celle que j'attendais le plus. De ce que j'avais déjà vu de LINQ et de ce qui tournait autour de cette nouveauté du framework 3.5 je savais qu'il y avait là un joyau... Les quelques essais que j'avais faits avec la Bêta de VS 2008 n'avaient qu'attisé ma curiosité. Je désirais donc à tout prix voir la session *DAT 201 :*

Entity Framework Introduction pour refaire le tour complet de la technologie pour être sûr de ne pas passer à côté d'un élément essentiel. [je parle ici des TechEd de 2007 qui se tenaient à Barcelone, cela ne nous rajeunit pas, mais il est parfois bon de revenir aux sources pour comprendre une technologie !].

Et c'est encore mieux que ce que je pensais. C'est une pure merveille, **une avancée aussi spectaculaire que la naissance du framework .NET lui-même**. Rien de moins.

L'Entity Framework

La session DAT 201, présentée par Carl Perry, Senior Program Manager Lead chez Microsoft, se voulait une introduction, un tour du propriétaire de l'Entity Framework. Mais kesako ? Le "*cadre de travail pour entité*" en français. Avec ça vous êtes bien avancé !

Pour faire le plus court possible disons que l'EF (Entity Framework) est une nouvelle couche d'abstraction se posant sur ADO.NET et permettant tout simplement d'accéder aux données de façon totalement objet et surtout dans le respect d'un schéma conceptuel et non plus en travaillant avec la base de données, son schéma physique et son SQL.

Est-ce que vous êtes plus avancé ? j'ai comme un doute... Mais les choses vont s'éclaircir avec la suite de ce billet.

Pour mieux comprendre la différence ne serait-ce que dans le code, comparons les deux types d'accès aux données, ADO.NET 2.0 et EF.

L'accès aux données de ADO.NET 2.0

Voici un code typique d'accès aux données via ADO.NET 2.0 tel qu'on en voit dans toutes les applis ou presque :

```
// Create a connection with the AdventureWorks connection string.
using (SqlConnection conn = new SqlConnection(
    ConfigurationManager.ConnectionStrings["AdventureWorks"].ConnectionString))
{
    conn.Open();

    // Create a command to join Customer and CustomerContactInfo tables.
    SqlCommand command = conn.CreateCommand();
    command.CommandText = @"
        SELECT  cust.FirstName, cust.LastName, contact.EmailAddress
        FROM    [dbo].[Customer] AS cust
        JOIN    [dbo].[CustomerContactInfo] AS contact
        ON      cust.CustomerID = contact.CustomerID
        WHERE   cust.MiddleName IS NULL";

    // Execute the command and obtain a data reader.
    using (SqlDataReader reader = command.ExecuteReader(
        CommandBehavior.SequentialAccess))
    {
        while (reader.Read())
        {
            // Write a response line using values from the reader.
            Response.Write(String.Format("<p>{0}\t{1}\t{2}</p>",
                reader["FirstName"],
                reader["LastName"],
                reader["EmailAddress"]));
        }
    }
}
```

On retrouve toute la chaîne d'opérations usuelles, de la création et l'ouverture de la connexion jusqu'à l'itération dans le resultset retournée par le DataReader suite à l'envoi d'une Command dont le contenu est du pur SQL exprimé dans le dialect spécifique de la base de données cible.

Cette méthode est puissante, souple, mais elle possède de nombreux pièges. Par exemple l'accès aux données du DataReader s'effectue par les noms de champ. Une simple coquille à ce niveau et le bug ne sera détecté qu'à l'exécution, peut-être dans 6 mois chez le client quand il utilisera cette fonction particulière du logiciel. Le coût de cette coquille peut être énorme s'il s'agit d'un logiciel diffusé en grand nombre (hot line de maintenance, détection du bug - pas toujours simple, création d'une mise à jour - les sources elles-mêmes ont bougé, problème de versionning, diffusion de la mise à jour et découverte de nouveaux problèmes - incompatibilités d'une des modifications, etc, etc). J'arrête là le scénario catastrophe car selon les lois de murphy je suis encore en dessous de la réalité, et vous le savez comme moi...

Le pire c'est que je n'ai relevé qu'une seule des erreurs possibles, ce code possède bien d'autres pièges qui agiront comme des aimants à bugs ! Un autre exemple,

toujours sur l'accès aux données lui-même : Il n'y a aucun contrôle de type. Je passe sur les conséquences que vous pouvez imaginer dans certains cas.

Car il y a plus grave encore : pour obtenir les données il faut taper un ordre SQL spécifique à la base cible... D'une part cela n'est pas portable, d'autre part ce n'est que du texte pour le compilateur. On écrirait "coucou" à la place de l'ordre SELECT ça compilerait tout aussi bien.

Je ne vous refais pas l'énumération des problèmes en cas de coquille (fort probable) dans une longue chaîne SQL. Je vous épargnerais aussi la liste des ennuis si l'admin de la base de données change le nom d'un champ ou le type de l'un d'entre eux ou s'il réorganise une table en la coupant en 2 ou 3 autres pour des raisons d'optimisation. C'est tout le soft qui devra être corrigé à la main sans rien oublier.

Bref, aussi génial que soit le framework .NET, aussi puissant que soit ADO.NET, on en était encore au même style de programmation des accès aux données que ce que ADO Win32 ou dbExpress chez Borland avec Delphi permettaient. Une logique proche, collée devrais-je dire, au schéma physique de la base de données et totalement inféodée à son dialect SQL, le tout avec des accès non typés aux données.

Certes ADO.NET 2.0 permettait d'aller un cran plus loin avec la génération des DataSet typés sous VS 2005. Une réelle avancée, mais qui n'était qu'un "arrangement" avec le modèle physique.

L'accès aux données avec ADO.NET Entity Framework

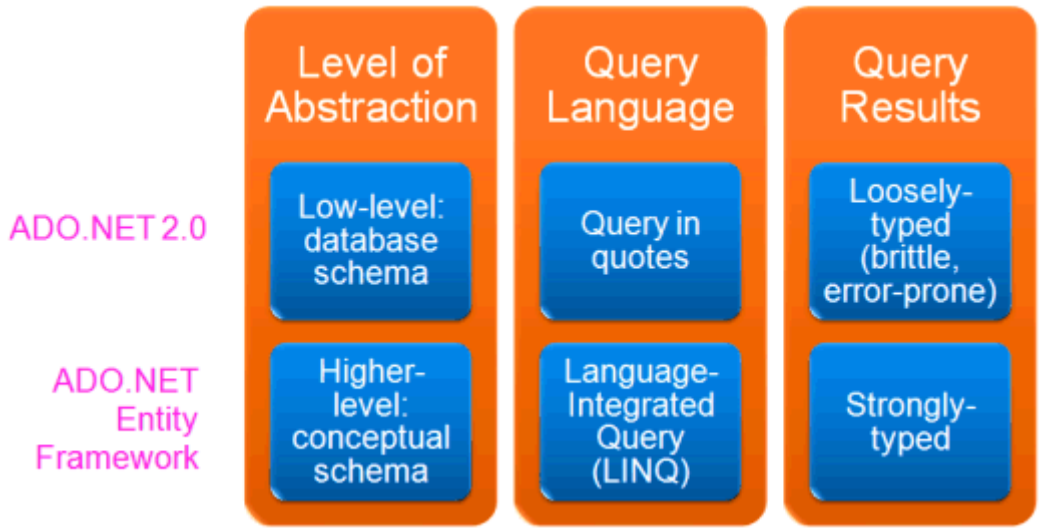
```
using (AdventureWorksModel model = new AdventureWorksModel())
{
    var query = from c in model.Customer
                where c.MiddleName == null
                select new {
                    FirstName = c.FirstName,
                    LastName = c.LastName,
                    EmailAddress = c.EmailAddress };

    foreach (var c in query)
    {
        Response.Write(String.Format("<p>{0}\t{1}\t{2}</p>",
            c.FirstName,
            c.LastName,
            c.EmailAddress));
    }
}
```

Dans cet exemple (qui fait exactement la même chose que le précédent) et dont nous comprendrons plus loin les détails on peut distinguer plusieurs choses :

1. Il n'y a plus de SQL, il a disparu.
2. On ne s'occupe plus de l'ouverture et de la fermeture de la connexion ni même de sa création.
3. Les données sont interrogées en C#, bénéficiant d'IntelliSense et du contrôle syntaxique.
4. On itère à travers une liste d'objets dont les propriétés sont typées.

Non, ne cherchez pas l'astuce de ce côté-là... si si, vous vous dîtes "le SQL a été tapé ailleurs il est simplement caché maintenant". Non. Dans cette façon d'accéder aux données personne n'a tapé de SQL, personne n'a créé de connexion à la base. Enfin, si, quelqu'un s'est chargé de tout cela de façon transparente : l'Entity Framework.

L'évolution dans les accès aux données



Sur le schéma ci-dessus on voit en colonne, respectivement : le niveau d'abstraction, le langage d'interrogation et le résultat des requêtes. En ligne nous trouvons ADO.NET 2.0 et, en dessous, ADO.NET EF.

- Le niveau d'abstraction de ADO.NET 2.0 est de type bas niveau, c'est à dire qu'il colle au schéma de la base de données (le modèle physique, le MPD).

Le niveau d'abstraction de EF change totalement puisqu'ici on travaille en haut niveau, celui du schéma conceptuel (le MCD).

- Côté langage d'interrogation, ADO.NET repose sur le dialecte SQL de la base cible exprimé sous la forme de chaînes de caractères dans le code.
  De son côté EF offre LINQ, ou Requêtage intégré au Langage (Language-Integrated Query) totalement indépendant de la base cible.
- Enfin, pour les résultats des requêtes, ADO.NET offre un accès non typé source d'erreurs, là où EF offre des objets aux propriétés typées.

On le voit ici clairement, le **bon technologique est faramineux**, c'est bien au niveau d'un modèle conceptuel qu'on va travailler en ignorant tout du modèle physique et du langage de la base de données !

C'est d'autant plus extraordinaire lorsque, comme moi, on commence à avoir quelques heures de vols et qu'on a connu moult tentatives couteuses et infructueuses chez plusieurs éditeurs de logiciels de créer des couches d'abstraction pour accéder aux données de différentes bases sans modifier l'application... Ce rêve de pouvoir fournir qui sa compatibilité, qui son logiciel de gestion médical sous SQL Server autant que Oracle ou MySQL en configurant juste l'application mais sans la recompiler ni même en maintenir plusieurs versions, ce rêve est aujourd'hui une réalité, une extension naturelle de .NET !

L'Entity Data Model (EDM)

L'EDM est le schéma conceptuel des données dans EF. C'est un vocabulaire qui décrit le schéma conceptuel pour être exact. Visual Studio 2008 le représente graphiquement sous forme d'entités et de liens comme un diagramme de classe.

Il permet au développeur de fixer le cadre des données qu'il souhaite voir, conceptuellement et sans rapport direct avec le schéma physique de la base de données.

EDM représentent d'une part les entités et des relations.

Les entités sont des types distincts (des classes) qui possèdent des propriétés qui peuvent être simples (scalaires) ou complexes. Les relations décrivent pour leur part la façon dont les entités sont liées. Il s'agit d'une déclaration explicite des noms de relation et des cardinalités.

Mais quand met-on les mains dans le camboui ? Il faut bien qu'à un moment ou un autre ce modèle conceptuel puisse correspondre avec le schéma de la base de données... Oui, bien entendu, mais une fois encore, pas de bas niveau ici.

Le schéma EDM peut être créé et modifié totalement graphiquement par le développeur (sous VS 2008) mais ce dernier possède aussi le moyen de faire le mapping entre les entités "idéales" de EDM et les entités physiques de la base de données. Pour cela deux façons coexistent : la plus simple consiste à faire un drag'n drop entre l'onglet des connexions SGDB de VS 2008 et l'espace de travail du schéma EDM ! On prend des tables, on les pose, et le schéma s'écrit de lui-même, relations comprises.

Par exemple le pattern courante "many to many" qui relie une table de lycéens à la table des options suivies par chacun fait intervenir une troisième table dans le schéma physique, celle qui contient les paires de clés lycéen/option. Ainsi un lycéen peut suivre plusieurs options et une option peut être suivie par plusieurs lycéens. Un classique des classiques. Ce pattern est détectée par EDM qui créé automatiquement une jointure n-n entre les deux entités, comme dans le schéma conceptuel de la base de données.

EDM automatise ainsi 95% de la création du schéma conceptuel, même si celui-ci ressemble encore beaucoup au modèle physique de la base. C'est là qu'interviennent les 5% restant : le développeur peut à sa guise supprimer ou ajouter des champs, des relations, et il indique explicitement à EDM comment les mapper sur les tables existantes dans la base de données.

Prenons l'exemple d'une base de données correctement optimisée et standardisée. Une fiche client pouvant posséder une adresse de livraison et une adresse de facturation, le concepteur de la base à séparer les deux informations. On a d'une part une table des clients et de l'autre une table des adresses plus deux jointures AdresseFacturation et AdresseLivraison qui parte de Client vers Adresse avec une cardinalité 0..1/1..1, c'est à dire qu'un client peut posséder 0 ou 1 adresse de facturation, 0 ou 1 adresse de livraison et que chaque adresse doit correspondre à 1 et 1 seul client (si on ajoute un type dans Adresse pour indiquer Facturation ou Livraison, le MCD pourra d'ailleurs exprimer la cardinalité sous la forme d'un lien dit identifiant avec Client).

Sous EDM, si on place les tables par drag'n drop, par force on trouvera dans le schéma les deux entités Client et Adresse avec ses liens. Toutefois cela n'est pas réellement intéressant. D'un point de vue conceptuel nous allons travailler sur des

fiches Client, et ses fiches possèdent une adresse de livraison et une autre de facturation. Pour nous, conceptuellement, ces adresses ne sont que des propriétés de Client. On se moque totalement des problèmes d'optimisation ou des règles de Codd que l'admin de la base de données a ou non suivi pour en arriver à créer deux tables... Cela appartient au modèle physique et nous ne voulons plus nous embarrasser de sa lourdeur et de son manque de pertinence conceptuelle.

Le développeur peut alors ajouter à Client deux propriétés, AdresseFacturation et AdresseLivraison, qui seront des propriétés complexes (retournant une fiche Adresse). Il supprimera aussi l'entité Adresse du schéma et enfin indiquera à EDM le mapping de ces deux nouveaux champs.

A partir de maintenant l'application pourra obtenir, trier, filtrer, des instances de Client qui possèdent, entre autres propriétés, une adresse de livraison et une autre de livraison. Plus aucun lien avec le schéma de la base de données, uniquement des choses qui ont un sens du point de vue conceptuel. C'est EF qui s'occupera de générer les requêtes sur les deux tables pour créer des entités Client.

Cela fonctionne aussi en insertion et en mise à jour... Le rêve est bien devenu réalité. Mieux, on peut personnaliser les requêtes SQL (en langage de la base cible) si vraiment on désire optimiser certains accès. De même on peut utiliser des procédures stockées au lieu de table ou de vues, etc.

L'aspect purement génial de EF c'est qu'il offre un mode automatique totalement transparent pour travailler au niveau conceptuel sans jamais perdre la possibilité, à chaque niveau de sa structure, de pouvoir intervenir manuellement. Qu'il s'agisse des entités, de leur mappings, des relations autant que la possibilité de saisir du code SQL de la base cible. Certes ce genre de choses n'est absolument pas à conseiller, en tout cas pour l'écriture manuelle de SQL, c'est un peu un contre-emploi de EF... Mais que la possibilité existe montre surtout l'intelligence de la construction de EF qui n'est pas une "boîte noire" sur laquelle il est impossible d'intervenir et qui, par force, limiterait les possibilités dans certains cas.

Interroger les données

Pour interroger les données, EF nous donne le choix entre deux méthodes :

LINQ to Entities

C'est l'exemple de code donné en début de ce billet. LINQ To Entities est une extension de C# qui possède des mots clés proches de SQL (from, where, select..). Mais soyons clairs : LINQ to Entities permet d'interroger le modèle conceptuel, l'EDM, et non pas la base de données ! C'est le mélange entre LINQ et EDM qui permet à EF de générer lui-même le code SQL nécessaire.

LINQ est d'une extraorinaire puissance. D'abord on reste en C#, on bénéficie donc de Intellisense, du contrôle de type, etc. Ensuite, LINQ to Entities n'est qu'une des émanations de LINQ puisqu'il en existe des variantes comme LINQ to XML qui permet globalement la même chose mais non plus en interrogeant des schémas EDM mais des données XML. Le fonctionnement de LINQ dépasse le cadre de ce billet et j'y reviendrai tellement il y a de choses à en dire.

Entity SQL

Si tout s'arrêtait là, Entity Framework serait déjà énorme. J'en perds un peu mes superlatifs tellement je trouve le concept et son implémentation aboutis. Je suis un fan de EF et de LINQ, vous l'avez remarqué et vous me le pardonnerez certainement une fois que vous l'aurez essayé...

Mais en fait EF propose un second moyen d'interroger les données, la tour de babel SQL, c'est à dire Entity SQL.

Il existe en effet de nombreux cas où LINQ pourrait s'avérer plus gênant que génial. Par exemple dans certains cas où le requêtage doit être dynamique (la requête est construite en fonction de choix de l'utilisateur, un écran de recherche multicritère par exemple). Décrocher de EF pour ces cas (pas si rares) imposerait de revenir à du SQL spécifique de la base cible, écrit dans des strings non testables, recevoir des données non typées, créer et gérer des connexions à la base en parallèle de LINQ utilisé ailleurs dans le code, et plein d'autres horreurs de ce genre.

Heureusement, Entity SQL existe. Il s'agit d'un langage SQL spécial, universel dirons-nous, qui ajoute la sémantique EDM à SQL. Ce SQL-là est spécial à plus d'un titre donc. Le premier c'est qu'il permet d'interroger l'EDM, le modèle conceptuel, comme LINQ, mais en SQL. Le second c'est que ce SQL "comprend l'objet" et sait utiliser non pas des tables et des champs mais des entités et des propriétés. Enfin, il est universel puisque lui aussi, comme LINQ, sera traduit par EF en "vrai" SQL de la base cible. Le même code Entity SQL fonctionne donc sans modification ni adaptation que la base soit Oracle, MySQL ou SQL Server !

Le rêve de certains éditeurs de logiciels dont je parlais plus haut s'arrêtait d'ailleurs là : créer un SQL interne à l'application qui serait traduit en SQL cible par un interpréteur lors de l'exécution. L'un des projets comme celui-là qui me reste en tête a coûté une fortune pour ne jamais marcher correctement... Entity SQL lui fonctionne de façon performante et s'intègre à Entity Framework ce qui permet de bénéficier de l'abstraction conceptuelle et de l'environnement objet. La réalité dépasse largement le rêve.

La traduction des requêtes

Toutes les requêtes sont exécutées par la base de données. Je le sous-entends depuis le début mais c'est mieux de le (re)dire pour qu'il n'y ait pas de méprise...

Les requêtes LINQ et Entity SQL sont transformées en requêtes SQL de la base cible par Entity Framework grâce à l'EDM qui contient la description des entités, des jointures et du mapping.

LINQ et eSQL fonctionnent sur le même pipeline de requêtag. Il transforme les constructions de EDM en tables, vues et procédures stockées. Ce socle commun est basé sur un modèle de fournisseurs qui autorise la traduction vers différents dialectes SQL (donc des bases cibles différentes).

EntityClient, le fournisseur d'accès aux données de EF

"N'en jetez plus !" pourraient s'écrier certains... Et oui, ce n'est pas fini. En effet, et comme je le soulignais, Entity Framework permet d'intervenir à tous les niveaux, du plus haut niveau d'abstraction au plus bas, dans la cohérence et sans pour autant être obligé de faire des choix ou se passer de tel ou tel avantage.

On a vu que par exemple on pouvait injecter du SQL cible dans le modèle EDM, même si cela n'est pas souhaitable, c'est faisable. On a vu qu'on pouvait préférer eSQL à LINQ dans certains cas. De même on peut vouloir totalement se passer de l'objectivation des entités pour simplement accéder aux données de façon "brutes" mais sans perdre tous les avantages de EF.

Pour ce cas bien particulier EF propose un fournisseur de données appelé EntityClient. Il faut le voir comme n'importe quel fournisseur de données ADO.NET. Il propose le même fonctionnement à base d'objets Connexion, Command et DataReader. On s'en sert comme on se sert d'un fournisseur SQL Server ou OleDB sous ADO.NET 2.0.

Toutefois, EntityClient interroge lui aussi le schéma conceptuel, l'EDM. Le langage naturel de ce provider d'accès est eSQL (Entity SQL) présenté plus haut.

Utiliser EntityClient peut s'avérer utile lorsque l'application n'a pas du tout besoin d'objets, ou lorsqu'elle possède sa propre couche objet (business object layer par exemple) qu'elle souhaite alimenter sans passer par les instances d'entités de EF tout en bénéficiant de l'abstraction de EF (utiliser un SQL universel notamment).

Les métadonnées

L'Entity Framework fournit un service d'accès aux métadonnées. Il s'agit d'une infrastructure commune permettant de décrire le stockage, le modèle, et la structure des objets, les procédures stockées, les informations de mapping et même les annotations personnalisées.

L'API publique permet de créer et d'utiliser des MetadataWorkspace directement, d'obtenir des MetadataWorkspace depuis des ObjectContext, d'utiliser les métadata retournées par les enregistrements de données de EntityClient.

Conclusion

La présentation de Carl Perry était entrecoupée de quelques démos permettant bien entendu de mieux voir et comprendre les concepts exposés. Visual Studio 2008 permet d'utiliser l'Entity Framework de façon simple et naturelle, le voir est toujours préférable à le croire sur parole. En tant que témoin de la session, et ayant "joué" avec la bêta de VS 2008 bien avant et étant depuis hier comme tous les abonnés MSDN l'heureux possesseur de la version finale de VS 2008, je peux vous certifier que tout ce que j'ai dit ici est.. en dessous de la vérité ! Testez vous-mêmes Entity Framework et vous comprendrez à quel point les temps ont changé, vous sentirez par vous-mêmes ce fantastique bond technologique qui relègue toutes les autres tentatives de ce type ou approchant (comme ECO de Borland par exemple) au rang des antiquités d'un autre millénaire... Entity Framework c'est une claque aussi forte que la sortie de .NET lui-même, on ne programmera jamais plus après comme on le faisait avant..

Entity Framework et la compatibilité arrière (problème Datetime2).

Vous possédez un OS récent, comme Windows 7, vous développez avec les outils les plus modernes (disons VS 2010 au hasard) et, bien entendu votre base de données installée en local est SQL Server 2008. D'une part parce que c'est une bonne version, mais surtout parce que vous n'avez pas trop le choix… Vous créez une application pour un client qui utilise SQL Server 2005 et là, pof! Dès qu'il exécute votre logiciel il y a une exception du genre "*The version of SQL Server in use does not support datatype 'datetime2'*"…

Vous cherchez alors comme un fou dans votre code, dans les métas donnés de la base où vous avez bien pu utiliser le type "`datetime2`". Mais rien. Que pouic. Néant.

La base de données n'utilise que des types SQL 2005, elle-même est bien une base SQL 2005, quant au code, vous en être certain, jamais aucune référence à Datetime2 nulle part.

Quel est ce mystère ?

Il se cache dans l'Entity Framework. Lorsque vous avez créé le modèle vous étiez connecté à votre serveur SQL Server 2008, peu importe que la base de données soit une base SQL 2005 ou non. C'est le serveur qui compte.

Or, le designer de l'Entity Framework optimise le modèle selon la version du serveur. C'est à dire que voyant que votre serveur est SQL 2008, il a stocké des informations permettant d'optimiser les requêtes pour cette version-là.

De fait, lorsque le logiciel est lancé, chez vous, ça passe impeccable… Mais chez le client ça bug. EF fabrique des requêtes et des classes à la volée, et il utilise notamment le type DateTime2 de SQL Server 2008 qu'il juge plus efficace que le type DateTime de la version précédente. Mais comme SQL Server 2005 ne sait pas traiter ce type-là, forcément ça coince.

La solution

Une fois qu'on sait c'est très simple.

Il suffit de charger le fichier .edmx (le modèle EF) dans un éditeur XML (ou le bloc-notes au pire) et de repérer en début de fichier généralement la balise suivante :

```
1: <Schema Namespace="Syo_DataModel.Store" Alias="Self"
2:   Provider="System.Data.SqlClient" ProviderManifestToken="2008"
3:   xmlns="http://schemas.microsoft.com/ado/2009/02/edm/ssdl">
```

Comme vous le constatez elle contient un attribut "`ProviderManifestToken`" qui, pour l'heure, indique la valeur "`2008`".

C'est simple : changer cette valeur en "`2005`" et dès lors le modèle sera "optimisé" pour SQL Server 2005. Cela continue de fonctionner sous SQL Server 2008 (si la base elle-même est bien au format 2005).

Conclusion

Une solution très simple pour une erreur qui peut prendre la tête. A noter dans vos tablettes donc !

Entity Framework : Include avec des jointures, inclure réellement les données

Entity Framework est d'une grande puissance mais c'est parfois une machine délicate. Le cas de Include avec une jointure en est un exemple vivant, l'Include semble ne pas fonctionner… Mais il y a une solution…

Eager Loading

Dans de nombreux cas il s'avère nécessaire de charger les entités liées à l'entité principale qui est requêtée, c'est le "eager loading" opposé au "lazy loading" qui diffère le chargement des grappes de données pour économiser bande passante et temps de traitement.

Voici par exemple une requête simple :

```
var results =
        from d in context.Dossiers.Include("Client")
```

```
        where d.Produit==”produit”
        select d;
```

Ici on imagine une entité “Dossier” et un ensemble de données “Dossiers”. Chaque Dossier contient une référence vers le client concerné. S’agissant d’un lien vers une autre entité (de type Client) les informations se trouvant dans le dossier sélectionné se limitent à l’ID du client par défaut.

Pour “remonter” la fiche client associée il faut avoir marqué les métadonnées de Dossier avec l’attribut d’inclusion, mais il faut aussi ajouter “Include(“Client”)” à la requête. La première action informe E.F. que l’inclusion est possible, la seconde la réclame effectivement.

Quand nous exécutons la requête ci-dessus nous obtenons bien une liste de Dossiers dont la propriété de navigation “Client” pointe réellement sur une fiche client utilisable.

Comme disent les américains, so far, so good…

Essayons maintenant de joindre la table des Dossiers à son historique dans la même requête :

```
var results =
        from d in context.Dossiers.Include(“Client”)
        join h in context.Historiques on d.IdDossier equals h.IdDossier
        where d.Produit==”produit” && h.DateDerniereAction>xxxxxxx
        select d;
```

Peu importe le sens de cette requête fictive donnée pour l’exemple, si vous la transposez à un cas réel chez vous, vous aurez la mauvaise surprise de voir, ou plutôt de ne plus rien voir de la fiche Client des dossiers retournés !

Pourtant l’Include est toujours là.

Quel est le problème ?

La jointure (qui pourrait aussi être exprimée par un simple second “from” et un “where”, le problème serait le même) change la forme de la requête, même temporairement…

Dans le premier exemple, ce sont des Dossiers qui sont sélectionnés, toutes les colonnes sont dans la requête, y compris celles concernées par l'Include. Du début à la fin du traitement de la requête seules des entités de type Dossier sont traitées, de fait les colonnes restent les mêmes. L'Include fonctionne.

Dans le second exemple quand l'Include est indiqué la requête inclut toutes les colonnes de Dossiers, mais la jointure (avec join ou un second from) modifie la forme de la requête, temporairement certes, mais cette modification impose de changer les colonnes puisqu'à ce moment la requête doit intégrer les colonnes des deux tables. Ce changement dans les colonnes traitées par la requête passe par un traitement qui '"oublie" l'Include de départ qui ne porte que sur une des deux tables. Include ne marche plus…

Solutions

Selon la requête, une solution consiste à réécrire une partie des conditions en utilisant des tests de type "any" ou autre groupage de données sur une sous requête qui n'est pas liée à la table qui possède des Includes. Je n'entrerai pas dans les détails de ces réécritures il en existe des dizaines de variantes selon les cas.

De plus cette solution est lourde car certaines requêtes ne seront pas évidentes à réécrire d'une autre façon.

L'astuce que je vous propose est bien plus simple car elle ne change rien à la requête :

```
var results =
        (from d in context.Dossiers // plus d'include
        join h in context.Historiques on d.IdDossier equals h.IdDossier
        where d.Produit==”produit” && h.DateDerniereAction>xxxxxxx
        select d).Include(“Client”);
```

Parfois la solution est si simple qu'on n'y pense pas… Entourez votre requête de parenthèses et ajoutez l'include à la fin. Et vous rétablirez le fonctionnement de ce dernier !

Attention toutefois : cela ne fonctionne que si la requête retourne une entité et non pas un "new { }" par exemple. Dans un tel cas il faudra traiter les données en deux

temps, remonter les dossiers avec leur client associés puis créer une nouvelle liste en mémoire ne contenant que les informations désirées. Cela peut être très couteux en espace mémoire et il faudra éventuellement ajouter une entité à votre schéma (de type POCO) pour ne retourner que les données nécessaires. Cela est surtout important lorsqu'on utilise Entity Framework avec les RIA Services ou d'autres services WCF.

Conclusion

Entity Framework est une machine sophistiquée, faire des requêtes simples avec Linq se démontre en quelques secondes dans une démo. S'en servir sur des cas réels révèle bien entendu des limitations ou des comportements qui ne sont plus aussi triviaux que dans une démo…

Heureusement la bête est assez souple et puissante pour trouver des contournements, encore faut-il connaitre ces derniers !

Modifier le Timeout des Wcf Ria Services

Les Wcf Ria Services sont très souples, ils permettent une large gamme de personnalisations, notamment celle du Tmeout des communications.

Timeout

Lorsqu'une application Silverlight fait un appel à un service Wcf Ria Services pour obtenir des données la communication qui s'établie est asynchrone. Je n'entrerais pas dans ces détails aujourd'hui mais il est nécessaire de le préciser. Car cela signifie qu'il n'y a pas vraiment de "tuyau" entre l'application et le serveur, en tout cas aucune communication permanente comme on pourrait le concevoir en client/serveur sur un réseau d'entreprise.

La tuyauterie existe bien, c'est WCF d'ailleurs, mais le plombier est un gars bizarre qui installe les tuyaux uniquement quand vous ouvrez le robinet pour demander de l'eau et qui démonte immédiatement la tuyauterie jusqu'à temps que la réserve d'eau renvoie de l'eau en réponse à votre demande, moment où il réinstalle (temporairement) tous les tuyaux pour que l'eau vous arrive, et ainsi de suite...

Un monde physique qui fonctionnerait de cette manière serait bien étrange... Mais en informatique rien n'est impossible puisque tout est virtuel !

De fait, lorsqu'une application Silverlight demande des données à un serveur exposant un service de type Wcf Ria Services il y a établissement d'une communication au moment de la demande puis plus grand chose jusqu'à ce que le serveur réponde, s'il répond.

Dans la réalité c'est un peu plus compliqué que cela, vous vous en doutiez, WCF est une mécanique assez complexe qui créée des canaux de communication restant virtuellement en place jusque ce que la réponse arrive. Mais WCF n'attend pas indéfiniment. Passé un certain délai il démonte vraiment la tuyauterie !

C'est à ce moment-là, si la réponse n'est pas arrivée bien sûr, que se déclenche une erreur de Timeout côté client (application Silverlight donc). Un dépassement de délai. Mais quel délai ?

Par défaut les WCF Ria Services utilisent un Timeout de 1 minute. Cela semble un choix très raisonnable. Toute requête qui dépasse ce temps est à revoir ou à séparer en plusieurs petites requêtes plus rapides. C'est une question d'efficacité.

Un petit plus long s'il vous plait M. Cadbury !

Bon, des requêtes plus longues qu'une minute c'est mal. Ok. Mais parfois on ne peut pas faire autrement. La requête ne peut pas être découpée et elle prend beaucoup de temps, on n'a pas la main sur le serveur Ria Services pour atomiser la requête en petites unités, etc...

Mais heureusement, il est possible de modifier le Timeout. Toute la question est de savoir où brancher la dérivation dans cette tuyauterie virtuelle plus complexe que les célèbres tuyaux d'aération du Centre Pompidou !

Vous me connaissez, je l'ai déjà dit, j'ai horreur de la plomberie. De la vraie, et pourtant je suis un bon bricoleur, autant que de la virtuelle (sécurité, protocoles de communications...). Tous ces machins me filent des boutons.

Je ne vous mentirai donc pas en vous disant que j'ai dépiauté sous Reflector le source de WCF pour trouver une solution, non, je l'ai repiquée à un plombier certifié, Kyle McClellan, un gars de l'équipe de développeurs des WCF Ria Services... Quitte à

recopier autant choisir ce qui se fait de mieux, c'est comme à l'école (il fallait être idiot pour se mettre à côté du dernier de la classe le jour d'une interro sur laquelle on ne se sentait pas très au point...).

Ce charmant développeur au patronyme qui fleure bon le vrai whisky pur malt ne doit pas abuser lui-même de la boisson de ses ancêtres écossais car sa solution est très lucide et assez simple (une fois qu'on sait comment faire, tout est là). Cela se décompose en deux étapes que je présente ci-dessous.

Une classe outil pour modifier le Timeout

C'est de la plomberie donc il faut des outils... La bête ne se livrera pas comme ça à vous sans un peu de code bien corsé. Donc dans un premier temps il faut déclarer dans un coin la méthode suivante :

```
 /// <summary>
/// Utility class for changing a domain context's WCF endpoint's
/// SendTimeout.
/// </summary>
public static class WcfTimeoutUtility
{
  /// <summary>
  /// Changes the WCF endpoint SendTimeout for the specified domain
  /// context.
  /// </summary>
  /// <param name="context">The domain context to modify.</param>
  /// <param name="sendTimeout">The new timeout value.</param>
  public static void ChangeWcfSendTimeout(DomainContext context,
                                          TimeSpan sendTimeout)
  {
    PropertyInfo channelFactoryProperty =
      context.DomainClient.GetType().GetProperty("ChannelFactory");
```

```
    if (channelFactoryProperty == null)
    {
      throw new InvalidOperationException(
        "There is no 'ChannelFactory' property on the DomainClient.");
    }

    ChannelFactory factory = (ChannelFactory)
      channelFactoryProperty.GetValue(context.DomainClient, null);
    factory.Endpoint.Binding.SendTimeout = sendTimeout;
  }
}
```

Ce n'est pas grand-chose, mais ça ne s'invente pas...

Deux options d'utilisation

La première consiste à appeler la méthode définie ci-dessus sur la variable de contexte avant que la moindre opération ne soit effectuée dessus. Je n'aime pas cette méthode car quel que soit l'endroit que vous choisirez il y aura toujours la possibilité qu'un petit malin (peut-être vous d'ailleurs) n'utilise le contexte ailleurs et avant dans une future modification du code...

Je préfère les méthodes sures et imparables.

La seconde méthode consiste plutôt à utiliser l'extensibilité prévue par Microsoft. Dans le code généré côté client (qui est caché par défaut dans VS mais que vous pouvez afficher en cliquant sur l'icône ad hoc du volet d'inspection du projet) les déclarations sont prévues en "partial" ce qui permet d'étendre le code sans créer de nouvelle classe. Et notamment la méthode OnCreate du contexte est déclarée en partial mais non utilisée par le code généré, elle est juste là pour qu'on puisse justement s'en servir dans un code partiel.

C'est donc par cette porte dérobée qu'il est préférable d'insérer l'appel à la méthode changement de Timeout.

Voici un modèle de code à adapter (selon les noms de classes et les namespaces) :

```
namespace SampleNamespace.Web.Services
{
  public partial class MyDomainContext
  {
    partial void OnCreated()
    {
      TimeSpan tenMinutes = new TimeSpan(0, 10, 0);
      WcfTimeoutUtility.ChangeWcfSendTimeout(this, tenMinutes);
    }
  }
}
```

Le problème est justement de remplacer les noms correctement sinon rien ne marchera, au mieux le code ne fera rien (il ne sera appelé par le contexte), au pire vous n'arriverez même pas à compiler.

D'abord ce code doit être déclaré côté client, dans l'application Silverlight donc. Pas sur le serveur.

Ensuite il faut modifier correctement les noms suivants :

"`SampleNamespace.Web.Services`" : c'est l'espace de nom dans lequel votre contexte est défini, si vous avez un doute, afficher les fichiers cachés du projet et dans le dossier "`Generated_code`" vous trouverez "`xxx.Web.g.cs`", le code source du proxy du service Ria que VS a créé pour vous.

"`MyDomainContext`" : c'est l'autre nom qui doit être modifié, il doit correspondre exactement au nom du contexte généré.

Dans l'exemple le Timeout est fixé à 10 minutes. C'est un exemple hein... Ne laissez pas la possibilité qu'un utilisateur puisse attendre dix minutes une requête, il aura zappé de site bien avant  . Et s'il s'agit d'une application d'entreprise, débranchez votre téléphone, jetez votre portable à la poubelle, changez de bâtiment et mettez

une fausse barbe (ou rasez celle que vous portez habituellement). Le port d'un casque de chantier me semble un plus pour votre sécurité.

Bien entendu, si une requête doit vraiment atteindre les dix minutes vous aurez tout intérêt à prévoir plus qu'une jolie animation...

Donc attention, prudence. La valeur par défaut d'une minute est déjà énorme...

Conclusion

Dépasser la minute de Timeout pour une requête est un choix que je ne ferai dans aucune application, sauf si on me le demande avec un gros chèque ou un colt Cobra sur la tempe (avis aux clients, je préfère la première solution ).

Toutefois dans certains cas il peut être intéressant de se protéger contre un Timeout intervenant rarement (quand les serveurs sont très chargés, quand le débit de la ligne est mauvais, etc).

A vous de voir. Maintenant vous savez comment faire...

WCF Ria Services : Charger les entités associées

WCF Ria Services est une mécanique de précision. En démo tout est toujours simple et évident, dans la réalité on rencontre toujours quelques cas plus retors et pas forcément dans des features ultra sophistiquées... Par exemple le chargement d'entités associées réclame de connaitre la double astuce que je vais vous présenter.

Des entités et des associations

De base, WCF Ria Services permet de publier des données relationnelles provenant d'un modèle Entity Framework. Aujourd'hui on peut aussi s'en servir avec des POCO (Plain Old CLR Object) mais restons dans le cadre de EF.

Imaginons une entité Customer, une classe associée à l'entité Activity. Cette dernière représente une activité enregistrée dans le dossier client, une sorte d'historique. Un client peut avoir de nombreuses activités, il s'agit, d'une association 1,1->0,n c'est à dire qu'un client peut avoir zéro ou n activités enregistrées alors qu'une activité est associée à un et un seul client de façon systématique.

Avec Entity Framework tout cela se fait automatiquement par reverse engineering de la base de données lorsqu'on créé le modèle.

Une telle relation (qu'on qualifie aussi de type "maitre /détail") est traversable dans les deux sens. Depuis l'entité Customer, EF met à notre disposition une propriété Activities (ou un autre nom choisi par le développeur) qui est une collection d'entités Activity. Depuis une entité Activity EF ajoute aussi une propriété de navigation "Customer" qui pointe vers l'entité client maitre.

Nous sommes vraiment là dans un cas rudimentaire ultra classique, ne cherchez pas où se trouve l'astuce pour le moment, on est dans le basique de chez basique !

Une fois le décor planté, passons aux choses sérieuses.

Interrogation des données avec les entités associées

Si j'interroge les clients (avec une requête LINQ depuis Silverlight par exemple) j'obtiendrais des entités clients. Chaque fiche contenant une propriété de navigation vers ses activités je m'attends à pouvoir travailler sur cette liste. Il n'en est rien. C'est l'aspect Lazy Loading des RIA Services.

Le Lazy Loading, littéralement "chargement paresseux", consiste à ne pas renvoyer toute la grappe des entités liées. C'est heureux car un système qui ne fonctionnerait pas comme cela serait inutilisable. Imaginez un instant que j'interroge une fiche client et que, automatiquement, le système me remonte l'ensemble de ses devis, factures, avoirs, avec chacun leur ligne article qui chacune renverrait la fiche fournisseur, etc... La requête d'une seule entité retournerait presque toute la base de données !

On doit pouvoir maitriser le chargement des entités associées pour éviter de tomber dans ce travers rédhibitoire. C'est ce que fait le Lazy Loading des Ria Services.

De même, en reprenant mon exemple Client->Activité, si j'interroge les activités j'aurais bien dans chacune une propriété de navigation "Customer", mais celle-ci sera "null" par défaut.

Prenons la requête suivante :

```
var q = from a in Activities where !a.HasBeenRead return a;
```

Il s'agit ici de remonter toutes les entités de type Activity dont la propriété HasBeenRead ("a été lue") est à False. De base il me sera impossible d'accéder à la ficher client de chaque entité Activity, bien que la propriété existe sa valeur sera nulle.

Autoriser le chargement des entités associées

La première chose à faire est d'indiquer lors de la requête qu'on souhaite charger les entités Customer associées à chaque entités Activity. La requête devient alors :

```
var a = from a in Activities.Include("Customer") where ...
```

"Customer" est la propriété pointant vers la fiche client de l'Activité. En ajoutant Include("propriété")nous demandons à LINQ et aux Ria Services d'inclure la fiche associée dans la grappe de données qui sera remontée.

On pourrait croire que cela est suffisant. Hélas non. La propriété Customer sera toujours retournée à null...

Pour que cela fonctionne il faut _aussi_ marquer de l'attribut Include la propriété Customer dans les métadonnées du modèle EF (dans la classe ActivityMetadata)...

```
 internal sealed class ActivityMetadata
{
   ...
   [Include]
    public Customer Customer { get; set; }
}
```

Et voilà... C'est tout bête mais il faut le savoir.

L'attribut Include dans les métadonnées autorise le chargement des entités associées, ce qui permet dans les requêtes LINQ d'utiliser la méthode Include lorsqu'on désire obtenir ces fameuses entités associées. Utiliser uniquement la syntaxe LINQ n'est pas suffisant si le chargement n'a pas été autorisé dans les métadonnées...

Conclusion

Rien de bien sorcier dans ce billet mais si on ne connait pas l'astuce on peut rester coincer un petit moment à se demander pourquoi les propriétés de navigation retournées par les requêtes restent obstinément à null alors même qu'on ajoute le Include dans les requêtes !

Ria Services, MVVM Light, Silverlight et données de conception (design time data) – Astuces

Lorsqu'on développe des applications Silverlight en utilisant plusieurs technologies à la fois comme les Wcf Ria Services et le toolkit MVVM Light, il peut y avoir des effets de bord imprévisibles. Notamment le non affichage des données de conception...

Tout marche bien et puis un jour, on ne sait pourquoi, les données de design ne s'affichent plus...

Pourtant MVVM Light prévoit dans le constructeur des ViewModels un test permettant de créer des données de conception en mode design (sous VS ou Blend). On vérifie, tout est ok.

On vérifie aussi que le DataContext de la page en cours est bien indiqué dans la balise d'ouverture du contrôle principal (page, UserControl...) car Blend et VS peuvent écrire cette liaison de deux façons, et la seconde (déclarée à part dans les ressources de l'objet) ne permet plus de bénéficier des dialogues de Binding sous VS... Astuce numéro 1 à se souvenir donc.

Mais malgré tout cela, les données de design continuent à se cacher !

Le plus souvent cela est dû au fait que le constructeur du ViewModel échoue lorsqu'il est exécuté par VS ou Blend au design. Vérifiez que le constructeur du ViewModel ne tombe pas en exception est la première chose à faire ici. Astuce numéro 2.

Il existe un cas particulier fréquent avec les Ria Services : chaque ViewModel déclare généralement son propre contexte. Et il est tentant d'écrire un code du genre :

```
private MonAppliDomainContext context = new MonAppliDomainContext();
```

Cela permet d'initialiser le contexte Ria directement dans la déclaration de la variable *context* utilisée ensuite par le ViewModel pour toutes les opérations Ria Services.

Or, la création d'un Domain Context échoue en mode Design. De fait l'instance du ViewModel échoue lors de sa création (par le biais d'une construction statique dans le ViewModel Locator de MVVM Light).

Ce n'est pas le constructeur de l'objet qui plante, mais sa construction... nuance subtile rendant le debug encore plus déroutant.

L'astuce numéro 3 consiste donc à utiliser correctement les facilités offertes par MVVM Light, notamment la détection du mode Design et d'instancier les variables de type Domain Context uniquement au Runtime :

```
public MonViewModel()

{

    if (!IsInDesignMode) context = new MonAppliDomainContext();


}


private MonAppliDomainContext context;
```

Bien entendu il ne faut pas que le code Design tente d'accéder au contexte, et bien sûr, cela s'étend à toute création d'instances qui ne sont pas directement utilisables au design time.

Blend et VS offrent d'autres solutions pour la création de données de Design, mais elles ne sont pas nécessaires si on utilise correctement les outils de MVVM Light. Mais j'en parlerai prochainement...

Une fois ces petits problèmes d'instanciation des ViewModels réglés, Blend et VS affichent de nouveau vos données de Design. Et la mise en page des applications s'en retrouve simplifiée.

Avertissements

L'ensemble des textes proposés ici sont issus du blog « Dot.Blog » écrit par Olivier Dahan et produit par la société E-Naxos.
Les billets ont été collectés fin septembre 2013 pour les regrouper par thème et les transformer en document PDF cela pour en rendre ainsi l'accès plus facile.
Les textes originaux ont été écrits entre 2007 et 2013, six longues années de présence de Dot.Blog sur le Web, lui-même suivant ses illustres prédécesseurs comme le Delphi Stargate qui était dédié au langage Delphi dans les années 90.

Ce recueil peut parfois poser le problème de parler au futur de choses qui appartiennent au passé… Mais l'exactitude technique et l'à propos des informations véhiculées par tous ces billets n'a pas de temps, tant que les technologies évoquées existeront …

Le lecteur excusera ces anachronismes de surface et prendra plaisir j'en suis certain à se concentrer sur le fond.

E-Naxos

E-Naxos est au départ une société éditrice de logiciels fondée par Olivier Dahan en 2001. Héritière d' Object Based System et de E.D.I.G. créées plus tôt (1984 pour cette dernière) elle s'est d'abord consacrée à l'édition de logiciels tels que la suite Hippocrate (gestion de cabinet médical et de cabinet de radiologie) puis d'autres produits comme par exemple MK Query Builder (requêteur visuel SQL).

Peu de temps après sa création E-Naxos s'est orientée vers le Conseil et l'Audit puis s'est ouverte à la Formation et au Développement au forfait. Faisant bénéficier à ses clients de sa longue expérience dans la conception de logiciels robustes, de la relation client, de la connaissance des utilisateurs et de l'art, car finalement c'en est un, de concevoir des logiciels à la pointe mais maintenables dans le temps.

C#, Xaml ont été les piliers de cette nouvelle direction et Olivier a été récompensé par Microsoft pour son travail au sein de la communauté des développeurs WPF et Silverlight. Toutefois sa première distinction a été d'être nommé MVP C#. On ne construit pas de beaux logiciels sans bien connaître le langage…
Aujourd'hui E-Naxos continue à proposer ses services de Conseil, Audit, Formation et Développement, toutes ces activités étant centrées autour des outils et langages Microsoft, de WPF à WinRT (Windows Store) en passant par Silverlight et Windows Phone.

A l'écoute du marché et offrant toujours un conseil éclairé à ses client, E-Naxos s'est aussi spécialisée dans le développement Cross-Plateforme, notamment dans le mariage des OS Microsoft avec Android, les deux incontournables du marché d'aujourd'hui et de demain.

N'hésitez pas à faire appel à E-Naxos, la compétence et l'expérience sont des denrées rares !