

Introduction à Entity Framework

Par Paul Musso 

Date de publication : 18 décembre 2008

Cet article est une introduction à Entity Framework, l'outil de mapping objet-relationnel de Microsoft.

Avant-Propos.....	3
Prérequis logiciels.....	4
1 - Concept d'Entity Framework.....	5
1.1 - Les entités.....	6
1.2 - Object Services.....	7
1.3 - Quels sont les SGBDR supportés ?.....	7
1.4 - L'éditeur de modèle EDM.....	8
2 - Conception d'un modèle EDM.....	10
2.1 - L'héritage.....	12
2.1.1 - Une table par type d'entité.....	13
2.1.2 - Une seule table pour toutes les entités.....	15
2.2 - Les procédures stockées.....	18
2.2.1 - Import de fonctions.....	18
2.2.2 - Création, mise à jour et suppression d'entités.....	19
2.3 - Une entité pour 2 tables.....	21
2.4 - Les types complexes.....	23
3 - Utilisation des entités et du contexte.....	25
3.1 - Opérations CRUD.....	25
3.1.1 - Lecture des données.....	25
3.1.2 - Création, mise à jour et suppression.....	27
3.2 - Personnalisation.....	28
3.2.1 - Les classes entités.....	28
3.2.2 - La classe de contexte.....	29
3.3 - Sérialisation.....	29
3.4 - Relations avec le contexte.....	30
3.5 - Accès concurrentiel.....	31
3.6 - Transactions.....	32
Conclusion.....	34
Remerciements.....	35

Avant-Propos

Qui ne s'est jamais arraché les cheveux pour récupérer ou mettre à jour des données stockées dans une base de données relationnelle comme SQL Serveur, Oracle ou MySQL ?

Ecrire des lignes de SQL directement dans votre code C# ou VB.Net peut s'avérer délicat et long à déboguer. Avec l'arrivée de LinQ To SQL, Microsoft propose une solution élégante pour requêter une base de données sans écrire une seule ligne de SQL. Les requêtes sont écrites avec des mots-clés du langage C# ou VB.Net et des objets, ceci permet une vérification de la syntaxe à la compilation.

Alors que LinQ to SQL propose seulement un mapping "Une classe = Une table", la nouvelle solution de mapping objet-relationnel de Microsoft, nommée Entity Framework propose une approche bien plus avancée.

Entity Framework est un outil permettant de créer une couche d'accès aux données (DAL pour Data Access Layer) liée à une base de données relationnelle. Il propose la création d'un schéma conceptuel composé d'entités qui permettent la manipulation d'une source de données, sans écrire une seule ligne de SQL, grâce à LinQ To Entities. Comparé à d'autres solutions de mapping objet-relationnel (ORM), Entity Framework assure l'indépendance du schéma conceptuel (entités ou objets) du schéma logique de la base de données, c'est-à-dire des tables. Ainsi, le code produit et le modèle conceptuel ne sont pas couplés à une base de données spécifique.

Cet article présente Entity Framework à travers les 3 aspects suivants :

- Les concepts d'Entity Framework
- Définition d'un modèle EDM
- Utilisation des entités et du contexte

La 1ère partie est une introduction théorique à l'outil. Alors que les 2 dernières concernent son utilisation par un exemple concret.

Prérequis logiciels

Pour utiliser Entity Framework et l'assistant de modèle EDM intégré à Visual Studio 2008, vous avez besoin des prérequis suivants :

- Visual Studio 2008 (édition standard, professionnelle ou team)
- Visual Studio 2008 Service Pack 1 (contient l'assistant de modèle EDM)
- SQL Serveur 2005 (toutes éditions confondues)

Le code sources de l'article (miroir [http](#)) contient 2 scripts SQL à installer qui se trouvent dans le dossier SQL, à la racine de l'archive :

- "DB-helloentityfx.sql" crée la base de données "helloentityfx", ses tables, vues et procédures stockées.
- "(FR)Data-helloentityfx.sql" (pour la version française de SQL Serveur 2005) et "(EN)Data-helloentityfx.sql" (pour la version anglaise de SQL Serveur 2005) insèrent les données de test dans la base "helloentityfx".

L'archive de l'article contient aussi un projet Visual Studio 2008 reprenant l'ensemble du code C# ainsi que le modèle EDM de l'article.



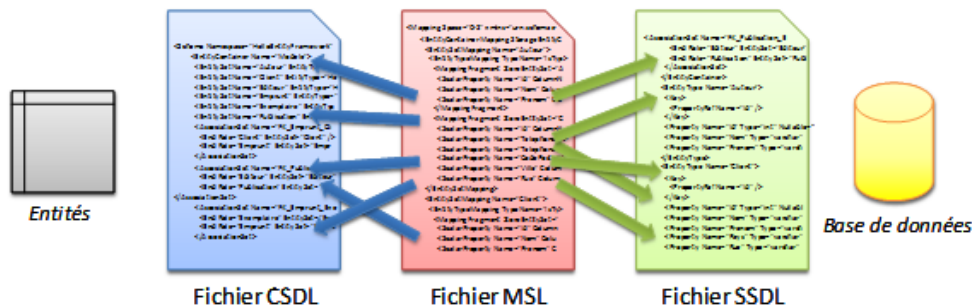
L'article fait référence à Visual Studio 2008 dans sa version anglaise. Toutes les démonstrations et explications sont aussi valables pour la version française du logiciel.

1 - Concept d'Entity Framework

Comme spécifié dans l'introduction, Entity Framework est la solution de mapping objet-relationnel proposée par Microsoft. Son but est de fournir la couche d'abstraction nécessaire aux développeurs pour qu'ils n'accèdent plus directement à la base de données, mais par l'intermédiaire d'entités définies par un modèle appelé EDM (Entity Data Model).

Ce modèle est divisé en 3 parties :

- Le schéma conceptuel : Il regroupe la définition des entités, des ensembles d'entités et des fonctions utilisées. Ces éléments sont définis dans un fichier XML appelé CSDL (Conceptual Schema Definition Language).
- Le schéma logique : Celui-ci correspond à la définition des tables, vues et procédures stockées déclarées dans la base de données. Toutes ces informations sont regroupées dans un fichier XML appelé SSDL (Store Schema Definition Language).
- Schéma de liaison : Il définit les liaisons entre le schéma conceptuel et le schéma logique. Il associe entre autres les entités déclarées dans le fichier CSDL aux tables définies dans le fichier SSDL. Ce mapping est inscrit dans le fichier XML MSL (Mapping Specification Language).



Cette architecture particulière permet une totale indépendance entre les objets utilisés dans la couche d'accès aux données et les éléments (tables, vues, procédures stockées) de la base de données. Il est ainsi possible de modifier le schéma logique sans que cela n'ait d'impact sur la définition des objets et inversement. Il est juste nécessaire de mettre à jour le schéma de liaison MSL.

Lors du traitement d'opérations CRUD (création, lecture, mise à jour et suppression), le framework utilise le modèle EDM pour construire les requêtes SQL. A partir d'une entité définie dans le fichier CSDL, il est possible de retrouver la ou les tables associées décrites dans le schéma logique grâce au schéma de liaison qui associe les entités aux tables.

Il existe 2 outils permettant de générer un modèle EDM : "EdmGen.exe" et l'assistant de modèle EDM intégré à Visual Studio. Le premier est contenu dans framework .Net 3.5. Celui-ci fonctionne en ligne de commande. Il permet entre autre de :

- Générer les fichiers CSDL, MSL, SSDL à partir d'une base de données.
- Générer les classes d'entité et la classe de contexte à partir du fichier CSDL.
- Valider le modèle EDM à partir de la base de données.


Pour plus d'informations sur l'outil "EdmGen.exe", veuillez vous référer à la [documentation officielle](#).

Visual Studio 2008 SP1 intègre aussi un assistant permettant de générer un modèle EDM, et ce de manière graphique. Il ressemble assez à l'assistant LinQ to SQL pour les connaisseurs et il offre un confort d'utilisation bien meilleur que l'outil en ligne de commande, au détriment d'un certain manque de fonctionnalités.

Par exemple, il n'est pas possible de définir de type complexe avec l'assistant, alors que "EdmGen.exe" le permet. Rassurez-vous, vous pouvez commencer à utiliser l'assistant (gain de temps évident) et utiliser par la suite l'outil

en ligne de commande. Néanmoins, il s'avère regrettable d'avoir recours à ce genre de manipulation. La prochaine version de l'assistant devrait corriger cela, et bien d'autres points.

L'article se base sur l'assistant de modèle EDM car il couvre la majorité des besoins utilisateurs.

 *L'assistant de Visual Studio 2008 concatène les 3 schémas (CSDL, MSL et SSDL) en un seul fichier de type "edmx". Ce fichier contient aussi des informations relatives à l'agencement graphique des entités dans le modèle. Pour utiliser par la suite "EdmGen.exe", il faudra extraire manuellement les éléments XML des 3 schémas et les insérer dans 3 fichiers distincts.*

1.1 - Les entités

Qu'est ce qu'une entité ?

Par définition, c'est un concept abstrait définissant un ensemble logique de propriétés et ayant des relations d'association avec d'autres entités. Les entités sont définies dans le schéma conceptuel du modèle EDM. Elles sont matérialisées avec l'outil "EdmGen.exe" ou l'assistant par des classes héritant de "EntityObject" et faisant partie de l'espace de nom "System.Data.Objects.DataClasses".

Contrairement à des outils de mapping objet-relationnel classiques, le mapping vers la base de données ne peut se faire qu'à partir d'entités. Cette contrainte permet d'optimiser les performances et facilite la création des objets métiers. De plus, les classes générées sont facilement extensibles puisqu'elles sont partielles (mot clé "partial" dans la définition de la classe).

Afin de requêter la source de données, Entity Framework offre les 3 possibilités suivantes :

- LinQ to Entities : Support de LinQ vers les entités. C'est la méthode la plus communément utilisée. Elle offre l'avantage de la vérification des requêtes à la compilation et du support de l'auto-complétion.
- Entity SQL : C'est un langage ayant une syntaxe proche du SQL. Voici un [lien](#) le présentant plus en détail.
- Méthodes du générateur de requêtes : Ces méthodes permettent de créer des requêtes de manière fonctionnelle. Plus d'information sur le [lien](#) suivant.

Cet article se focalisera quasi exclusivement sur LinQ To Entities.

Peut-on utiliser des classes métier n'héritant pas de "EntityObject" ?

La réponse est oui ! Au minimum, il est nécessaire que vos classes héritent des interfaces suivantes :

- IEntityWithChangeTracker : Active le suivi des modifications (obligatoire).
- IEntityWithKey. Facultatif : Expose une clé d'entité (obligatoire).
- IEntityWithRelationships : Obligatoire pour les entités présentant des associations (facultatif).

Pour plus d'informations, voici la [documentation officielle](#).

Implémenter ces 3 interfaces sous-entend que vos classes métiers doivent être modifiées pour utiliser la persistance avec Entity Framework. En d'autres termes, le principe de faible couplage / forte cohérence n'est pas respecté puisque les classes métiers contiennent à la fois de la logique métier et de la logique de persistance. Un design POCO signifie que les classes métiers ne contiennent aucune logique de persistance. Finalement dans certains scénarios, il n'est pas possible de modifier ses classes métiers.

On peut en tirer la conclusion suivante : Entity Framework ne permet pas d'utiliser des classes POCO (Plain Old Code/csharp Object). D'autres outils le permettent comme NHibernate avec l'utilisation d'objets proxy. Danny Simmons,

développeur de l'équipe d'Entity Framework s'explique à [ce sujet](#). Souhaitons que ce manque soit comblé dans les prochaines versions de l'outil.

Cependant, sachez qu'il est possible de contourner ce manque avec l'utilisation de la programmation orientée aspect. Sans rentrer dans les détails, [PostSharp](#) permet d'injecter du code lors d'une 2ème phase de compilation et donc d'implémenter les 3 interfaces nécessaires sur les classes POCO. [Un projet](#) a d'ailleurs été réalisé à ce sujet. Vous trouverez plus d'information sur [son blog](#).

1.2 - Object Services

Comment les entités communiquent avec Entity Framework ?

Toutes requêtes LinQ to Entities ou utilisant le générateur de requête s'appuient sur une couche appelée "Object Services". Cette couche logique permet de gérer les opérations de type CRUD sur les entités, de suivre les modifications des objets, de gérer l'accès concurrentiel et permet la liaison des entités avec des contrôles d'interface (data binding). Dans un sens, elle matérialise le résultat des requêtes en entités, et dans l'autre, elle transforme les requêtes émises en arbres d'expression vers une couche plus basse appelée "EntityClient".

La couche "Object Services" est principalement matérialisée par la classe "ObjectContext". Celle-ci est d'ailleurs dérivée lors de la génération des entités en classes. L'outil "EdmGen.exe" fourni par le framework .Net 3.5 génère en plus des entités définies dans le schéma conceptuel (CSDL) une classe appelée communément le contexte, qui dérive de "ObjectContext". Celle-ci est utilisée pour ouvrir la connexion, requêter, mettre à jour la base de données et bien plus. Tout comme les classes d'entité, la classe de contexte est elle aussi partielle, ce qui permet de l'enrichir fonctionnellement. Voici un exemple ouvrant une connexion et effectuant une requête de type SELECT :

```
using (Modele bdd = new Modele())
{
    // Définition de la requête LinQ
    var requete = from publication in bdd.Publication
                  select publication;

    // Execution de la requête
    requete.ToList().ForEach(p => Console.WriteLine("Publication : ID = {0} | Titre = {1}", p.Id,
    p.Titre));
}
```

La requête LinQ utilise une propriété spécifique, "Publication", de la classe "Modele" renvoyant un objet de type "ObjectQuery<Publication>". C'est grâce à cet objet que l'on peut requêter la source de données. Ici la requête récupère l'ensemble des publications insérées dans la table "Publication" de la base de données. Le résultat est de type "IQueryable<Publication>", qui hérite de la classe "IEnumerable<T>". Ceci permet de différer l'exécution de la requête, grâce à son itérateur, qui n'est exécutée qu'à l'appel de la méthode "ToList" transformant en une liste de type "List<Publication>" le résultat de la requête.

1.3 - Quels sont les SGBDR supportés ?

Comme cité plus haut, la couche "Object Services" s'appuie sur le fournisseur de données "EntityClient" pour qu'il exécute les requêtes LinQ transformées au préalable en arbres d'expression. "EntityClient" a pour but d'accéder aux données définies dans le modèle EDM décrit par les 3 fichiers CSDL, MSL et SSDL. Il génère ensuite un arbre d'expression relatif aux données du schéma logique et l'envoi à un fournisseur de données spécialisées, afin que celui-ci transforme l'arbre en requêtes SQL.


Ce dernier fournisseur de données est spécifique à la base de données utilisée. Le fournisseur utilisé est défini par la chaîne de connexion utilisée par l'objet "EntityConnection", qui a la responsabilité d'ouvrir la connexion de la source de données.

Voici un exemple de chaîne de connexion, définie par l'assistant de Visual Studio 2008 :

```

metadata=res://*/modele.csd|res://*/modele.ssd|res://*/modele.msl;
provider=System.Data.SqlClient;provider
connection string="Data Source=TLS-LOG-PO-PMO\SQLEXPRESS;
Initial Catalog=helloentityfx;Integrated Security=True;MultipleActiveResultSets=True"
  
```

La 1ère partie concerne les métadonnées de la connexion. On y voit la référence aux 3 schémas du modèle EDM, et du fournisseur de données utilisé pour la base de données. Ici, c'est la classe "System.Data.SqlClient" qui permet de communiquer avec une base de type SQL Serveur 2000/2005/2008.

 *L'assistant de modèle EDM génère les 3 schémas du modèle EDM et les insère en tant que ressource dans l'assembly du projet. C'est pour cela que la chaîne de connexion fait référence à des fichiers ressources.*

Il est donc possible de spécifier un autre fournisseur de données pour utiliser, par exemple, une base Oracle, MySQL ou PostgreSQL. Voici une liste non exhaustive des fournisseurs de données compatibles avec Entity Framework :

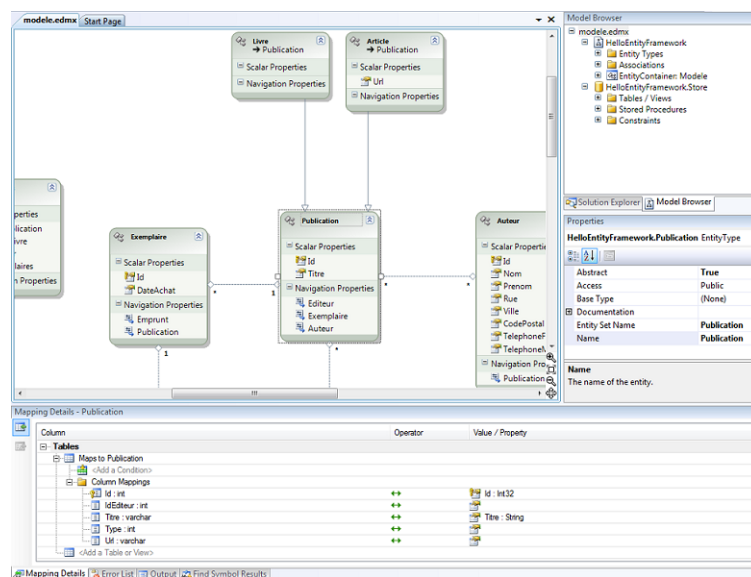
- Oracle : <http://devart.com/dotconnect/oracle/> (payant)
- MySQL : <http://devart.com/dotconnect/mysql/> (payant)
- PostgreSQL : http://pgfoundry.org/frs/shownotes.php?release_id=1230 (gratuit)
- SQLite : <http://sqlite.phxsoftware.com/>, <http://devart.com/dotconnect/sqlite/> (gratuit)

Il est aussi possible de créer son propre fournisseur de données. Un  **projet** sur Codeplex fournit le code nécessaire.

1.4 - L'éditeur de modèle EDM

L'éditeur de modèle EDM permet, tout comme "EdmGen.exe", de définir un modèle EDM, mais ici, de manière graphique. L'outil permet d'éditer les fichiers de type "edmx" stockant l'intégralité du modèle EDM (schémas CSDL, MSL, SSDL). Chaque sauvegarde du fichier "edmx" déclenche le custom tool "EntityModelCodeGenerator" générant les classes d'entité et le contexte spécifique.

Voici à quoi ressemble l'éditeur de modèle EDM :



L'éditeur est divisé en 4 parties :

- La fenêtre principale. Elle affiche les éléments du modèle EDM. Les entités sont représentées par une forme rectangulaire grise contenant les propriétés scalaires et les propriétés de navigation. Il est possible de rajouter des propriétés scalaires à partir du menu contextuel ("Add > Scalar Property"). Les associations entre entités sont représentées avec leurs cardinalités par un trait pointillé avec des losanges à chaque extrémité. Les relations d'héritage entre entités sont représentées par un trait plein et une flèche désignant l'entité mère.
- L'explorateur de modèle (en haut à droite) est un arbre où sont répertoriés les éléments du schéma conceptuel et ceux du schéma logique. Le menu contextuel (Update model from database) permet de mettre à jour le modèle EDM à partir de la base de données. Par exemple, si vous ajoutez une table, la mise à jour rajoute l'élément correspondant dans le schéma logique, ajoute une entité au schéma conceptuel et prédéfinit le mapping entre les deux.
- La fenêtre de propriétés (au milieu à droite) permet de définir les propriétés du modèle, des entités, des propriétés scalaires, de navigation et des associations sélectionnées.
- Les détails de mapping des entités (en bas). En sélectionnant une entité, le mapping, entité <-> base de données, est affiché. Il est possible de sélectionner la ou les tables en relation avec l'entité. Pour chaque table, vous pouvez définir la relation entre chaque colonne de la table et les propriétés de l'entité. L'éditeur de mapping permet aussi de spécifier les procédures stockées utilisées pour les opérations de création, de mise à jour, et de suppression d'une entité. Sachez qu'il n'est pas possible de définir qu'une procédure ou 2 sur les 3. Soit vous les spécifiez toutes, soit aucune.

La partie suivante de l'article montre comment utiliser l'éditeur afin de créer un modèle EDM pour la base de données "helloentityfx" (voir prérequis).

2 - Conception d'un modèle EDM

Cette partie a pour but de définir un modèle EDM grâce à l'assistant de Visual Studio 2008. Les points suivants sont traités :

- Création d'un modèle EDM avec l'assistant EDM
- Définition d'héritage
- Utilisation des procédures stockées
- Mapping d'une entité sur 2 tables
- Les types complexes

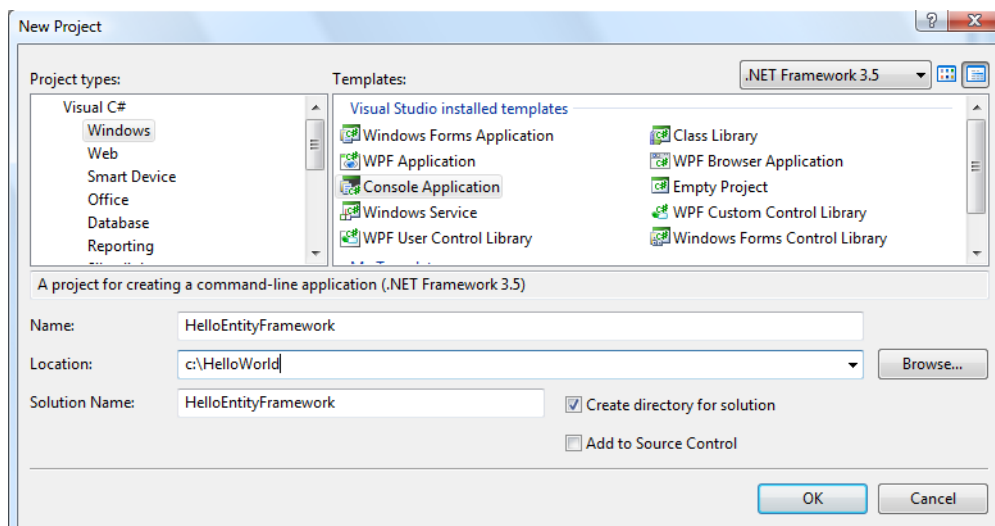
⚠ La suite de l'article nécessite d'avoir installé les 2 scripts SQL définissant et remplissant la base de données (voir les prérequis).

La base de données "helloentityfx" sert de source de données pour une application de gestion de livres et d'articles. Elle est composée de 11 tables :

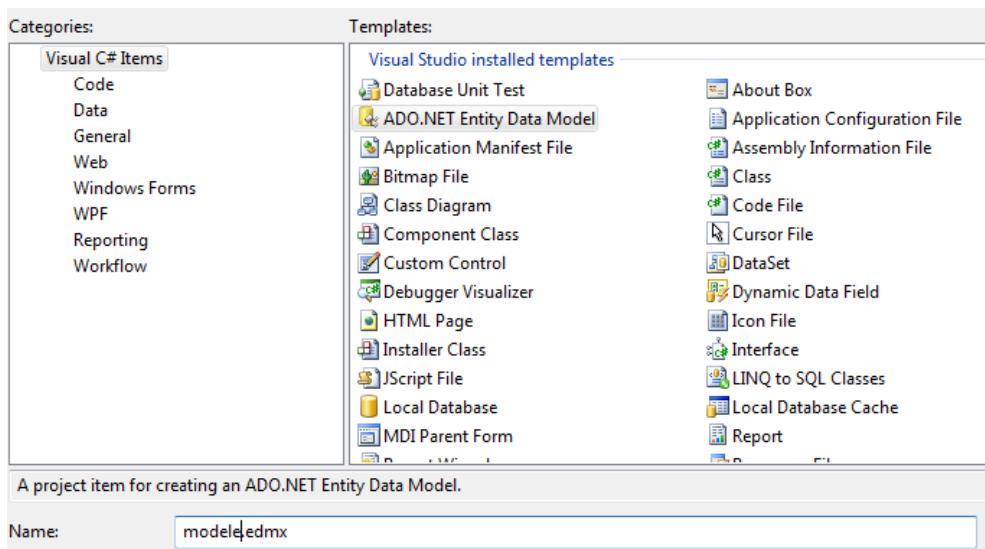
- Auteur
- CoordonneesAuteur
- Client
- Editeur, EditeurBD, EditeurInformatique, EditeurPsychologie
- Emprunt
- Exemple
- Publication
- Publi_Auteur

Le but de cette partie est de définir un modèle EDM et un schéma conceptuel permettant de mettre à jour et de requêter l'ensemble de ces tables par l'intermédiaire des entités ainsi que du contexte.

Pour commencer, créez un projet de type "Console Application", nommé "HelloEntityFramework":

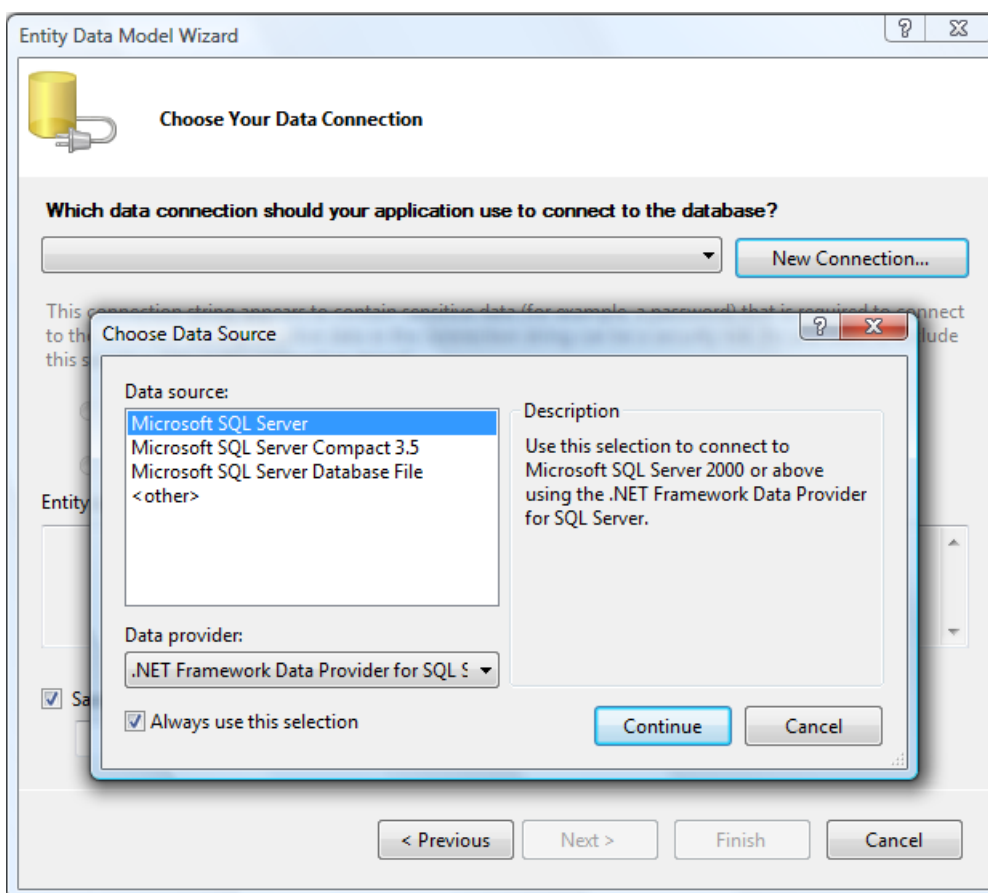


Ajoutez ensuite un nouvel item de type "ADO.Net Entity Data Model" au projet nommé "modele.edmx" :



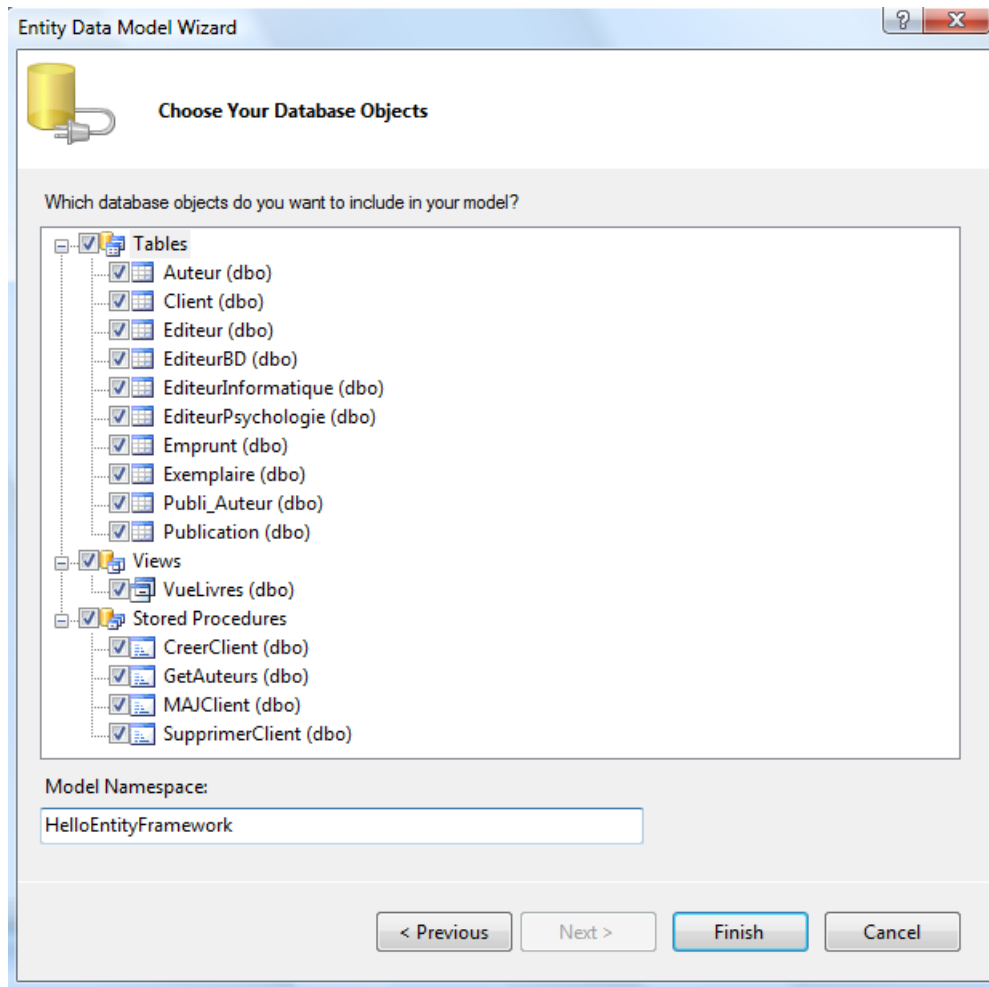
Sur la 1ère fenêtre de l'assistant, sélectionnez la 1ère icône, nommée "Generate from database". En faisant cela, les entités du schéma conceptuel vont être générées à partir de la base de données "helloentityfx". Cette opération va aussi créer le schéma logique SSDL (bases de données) ainsi que le mapping entre les entités et les éléments de la base de données (schéma MSL).

Ajoutez ensuite une nouvelle connexion de type "Microsoft SQL Server" :



Dans la fenêtre "Connection Properties", sélectionnez l'instance du serveur SQL Server hébergeant la base de données "helloentityfx" dans le champ "Server name" et choisissez la base de données "helloentityfx" dans la liste d'en-dessous. Validez ensuite la connexion aux données.

Il ne reste plus qu'à choisir les objets qui seront importés dans le modèle. Cochez "Tables", "Views" et "Stored Procedure", définissez le champ "Model Namespace" à "HelloEntityFramework" et cliquez sur "Finish" pour valider cette dernière étape.



Après validation, Visual Studio est appelle le custom tool "EntityModelCodeGenerator" transformant les données XML du fichier "modele.edmx" en un ensemble de classes dérivant de "System.Data.Objects.DataClasses.EntityObject" pour les entités et de "System.Data.Objects.ObjectContext" pour l'objet contexte.

La classe "contexte" est appelée par défaut "helloentityfxEntities1". Ouvrez le modèle avec l'éditeur de modèle EDM et dans les propriétés du modèle, définissez la valeur "Entity Container Name" à "Modele". Cette manipulation a pour effet de modifier le nom de la classe de contexte.

2.1 - L'héritage

Il existe 2 types d'héritage dans Entity Framework :

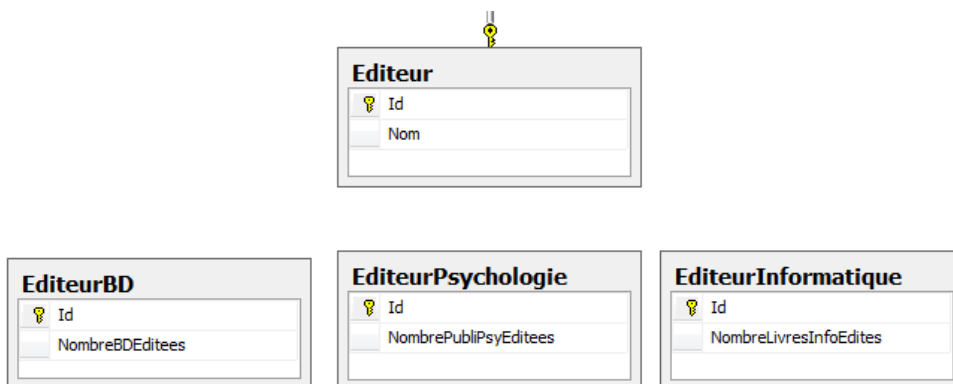
- **Une table par type d'entité** : Pour chaque entité dans la hiérarchie d'héritage, il existe une table dans la base de données.
- **Une seule table pour toutes les entités** : Quelque soit le nombre d'entités contenues dans la hiérarchie d'héritage, il n'existe qu'une seule table en base de données.

Le choix de l'un ou l'autre dépend en grande partie de la structure de la base de données.

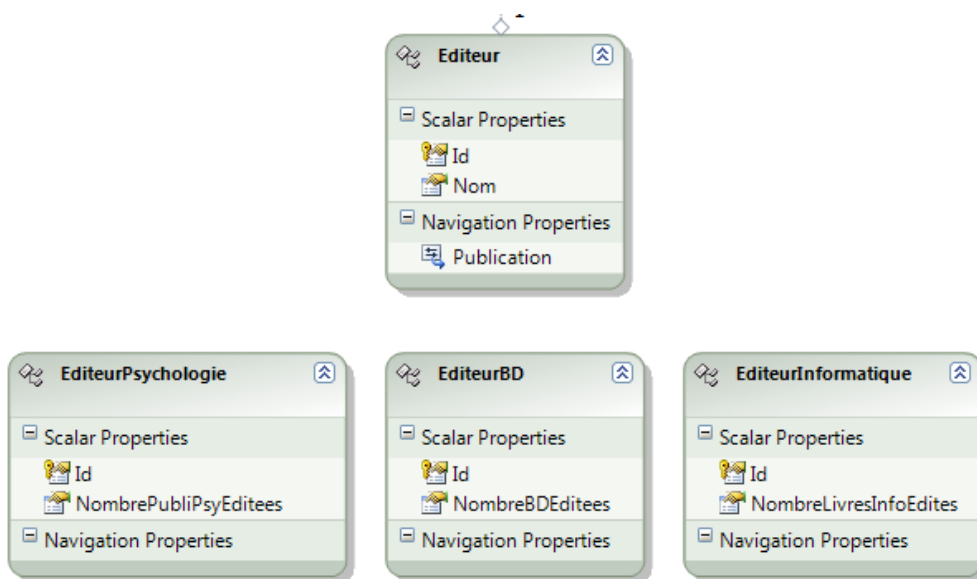
Nous allons voir comment implémenter ces 2 types d'héritage dans notre modèle.

2.1.1 - Une table par type d'entité

La base de données "helloentityfx" contient 4 tables appelées "Editeur", "EditeurBD", "EditeurPsychologie" et "EditeurInformatique" :

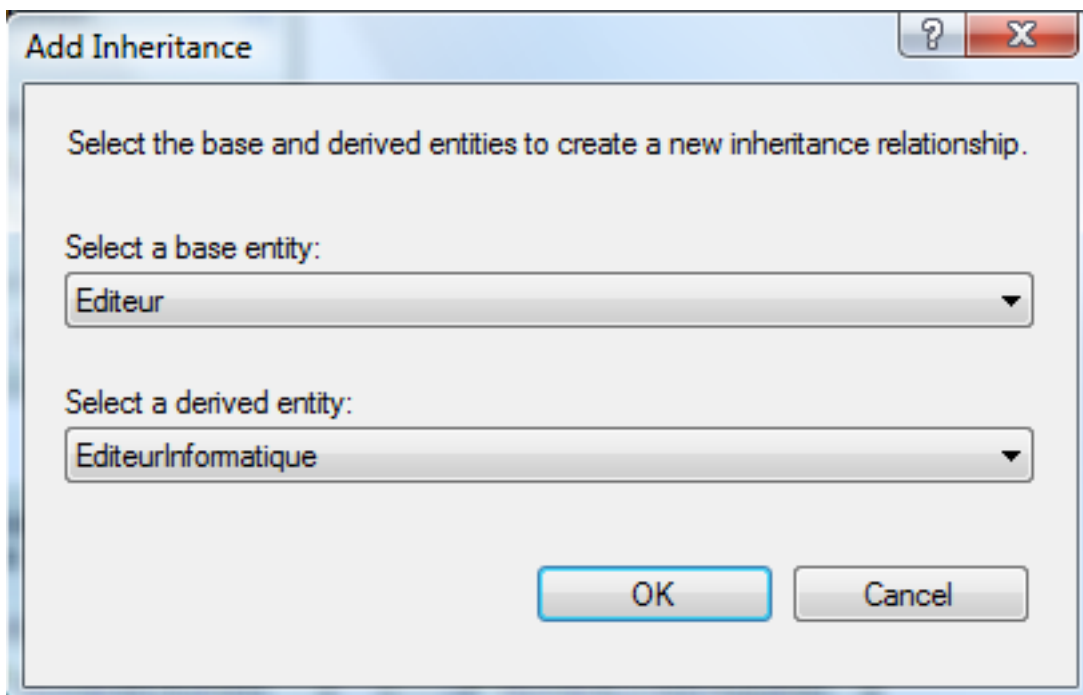


Du côté des entités, il y en a aussi 4 bien distinctes et n'ayant aucune relation d'héritage entre elles. Chacune correspond à une des 4 tables citées précédemment. Une bonne modélisation voudrait que les entités "EditeurBD", "EditeurPsychologie" et "EditeurInformatique" héritent de l'entité "Editeur" :



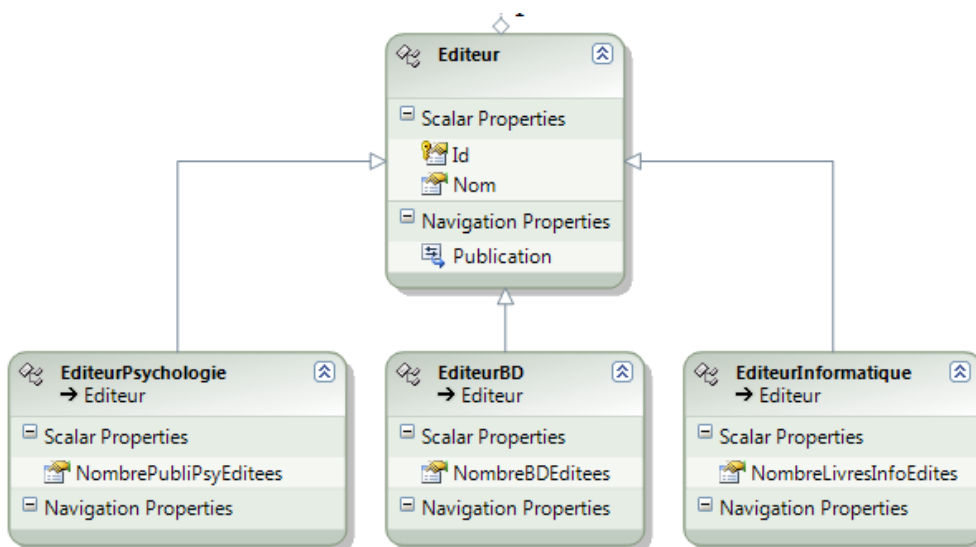
Pour cela, il suffit de répéter la manipulation suivante sur les 3 entités à spécialiser :

- Supprimez la propriété faisant office de clé d'entité (propriété Id dans notre cas) dans l'entité fille.
- Faites un click n'importe où dans l'éditeur de modèle EDM et sélectionnez "Add > Inheritance ...". Dans la liste déroulante "Select a base type", choisissez "Editeur" et dans l'entité à dériver, sélectionnez l'une des 3 entités citées précédemment. Validez le formulaire pour finir.



- Affichez les propriétés de mapping de l'entité dérivée, et définissez pour la colonne Id la propriété "Id:Int32".

Voici le résultat en image :



Pour chaque entité, une propriété de type "ObjectQuery<TypeEntité>", aussi appelée "Entity Set", est créée dans l'objet contexte. Cependant, pour les entités dérivées ce n'est pas le cas. Pour effectuer une requête de type SELECT concernant uniquement une entité spécifique, une bonne solution est d'utiliser la méthode "OfType<T>", voici un exemple :

```
using (Modele bdd = new Modele())
{
    var requete = from editeurInfo in bdd.Editeur.OfType<EditeurInformatique>()
                  select editeurInfo;
}
```

Cette requête récupère l'ensemble des éditeurs informatiques stockés dans les tables "Editeur" et "EditeurInformatique".

Pour l'insertion d'une nouvelle entité dérivée en base, il suffit d'appeler une méthode de l'objet contexte nommée ainsi : `AddTo[TypeDeBase]`.

```

using (Modele bdd = new Modele())
{
    EditeurInformatique OReilly = new EditeurInformatique()
    {
        Nom = "OReilly",
        NombreLivresInfoEdites = 1800
    };

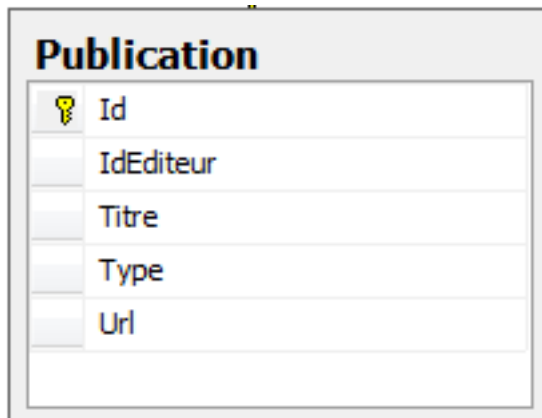
    bdd.AddToEditeur(OReilly);
    bdd.SaveChanges();
}


```

Du côté de la base de données, une nouvelle ligne est insérée dans la table "Editeur" ainsi que dans la table "EditeurInformatique". Les 2 éléments ont la même valeur de clé primaire.

2.1.2 - Une seule table pour toutes les entités

La base de données "helloentityfx" contient une table appelée "Publication". Celle-ci contient un champ "Type" de type int :

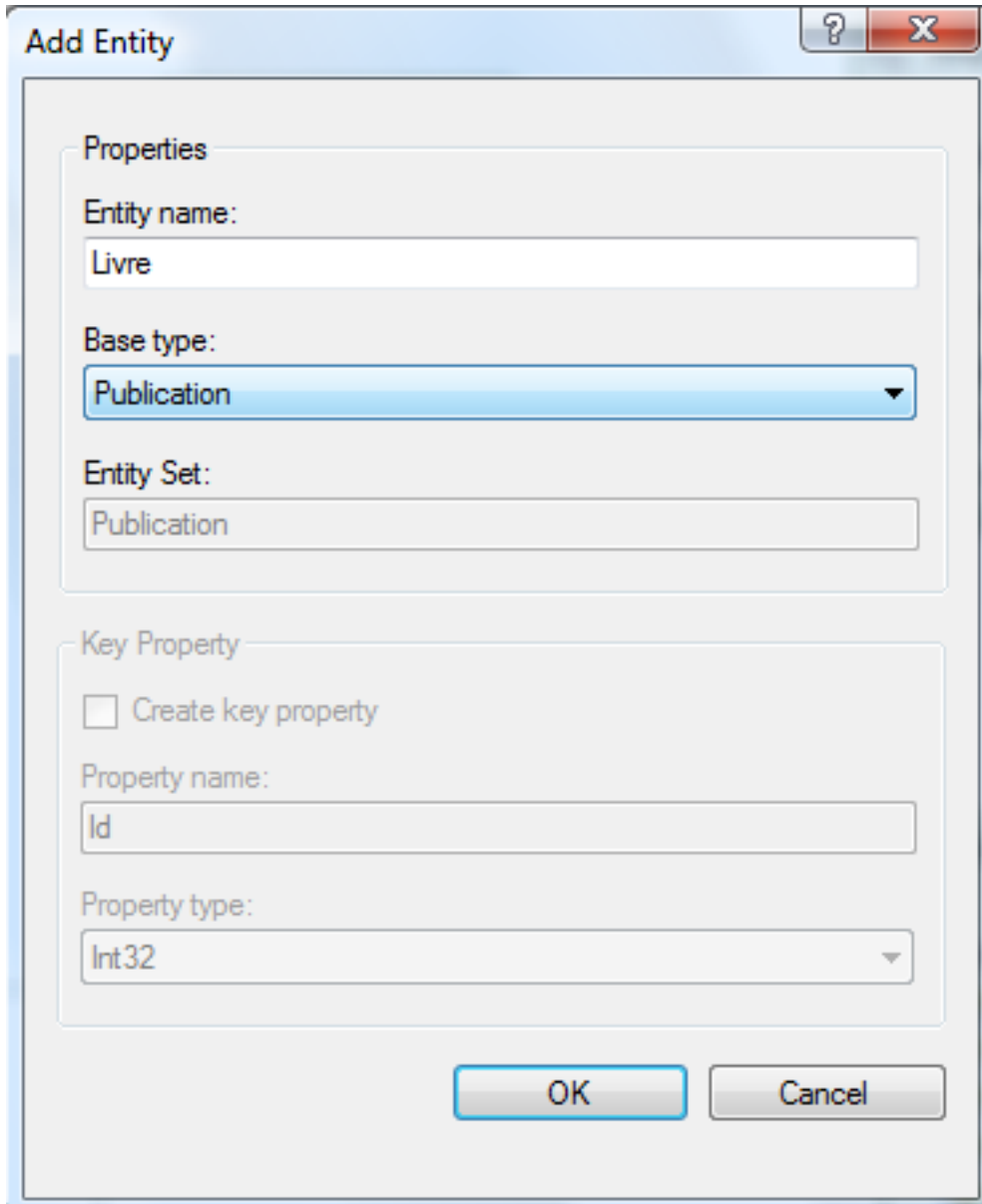


Publication	
	Id
	IdEditeur
	Titre
	Type
	Url

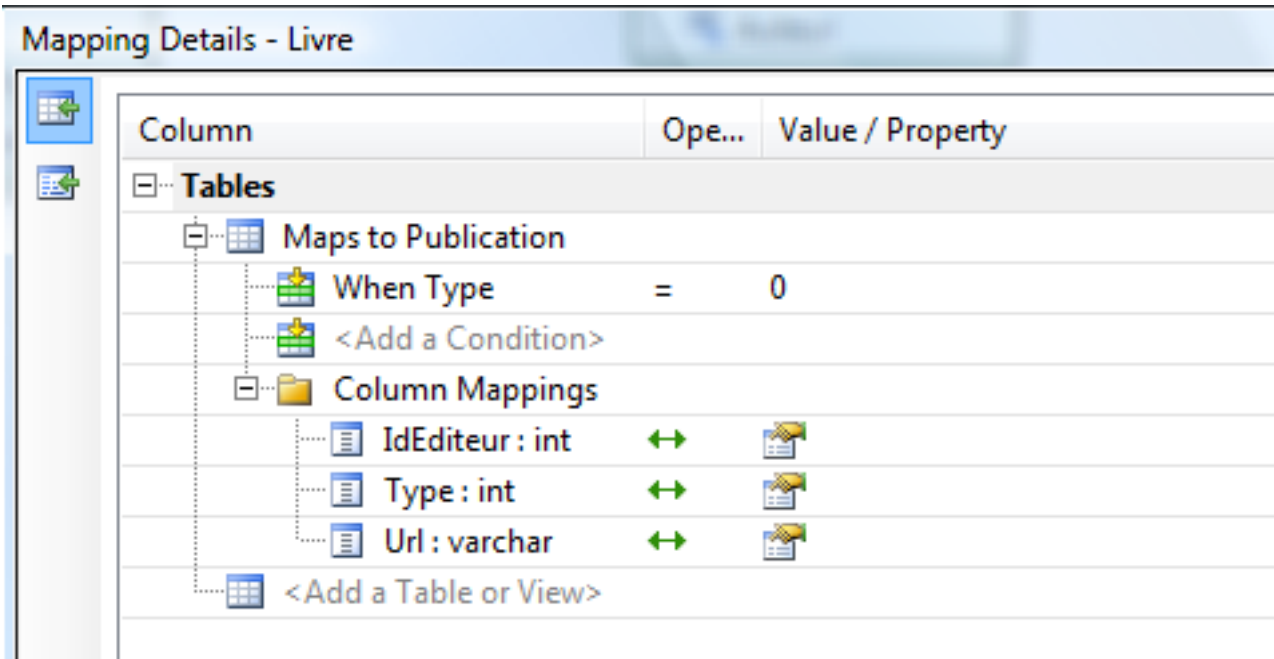
Au niveau du modèle, il existe une entité "Publication" mappée à la table du même nom. Le but est d'avoir 2 types de publication : les livres et les articles. Ces 2 entités doivent hériter de l'entité Publication. L'entité "Article" a la particularité d'avoir une propriété "Url", que n'a pas l'entité "Livre". Dernière précision, l'entité "Publication" n'a pas vraiment de sens en soi, le mieux est de la définir comme abstraite.

Pour commencer, supprimez la propriété "Url" et "Type" de l'entité "Publication". La 1ère sera définie uniquement dans l'entité "Article". Quand à la seconde, elle sert de condition pour l'héritage spécifié dans le schéma conceptuel. Dans les propriétés de "Publication", définissez l'entité comme abstraite (`Abstract = True`).

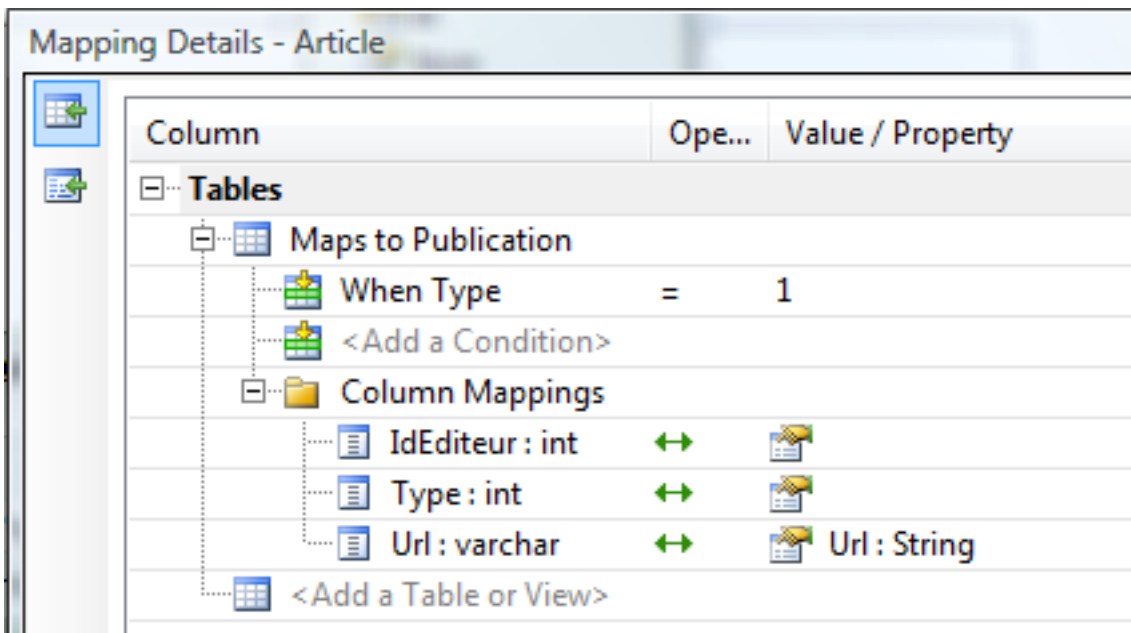
Afin de créer l'entité "Livre", faites un click droit quelque part dans l'éditeur de modèle EDM et choisissez "Add > Entity ...". Définissez les valeurs comme ci-dessous :



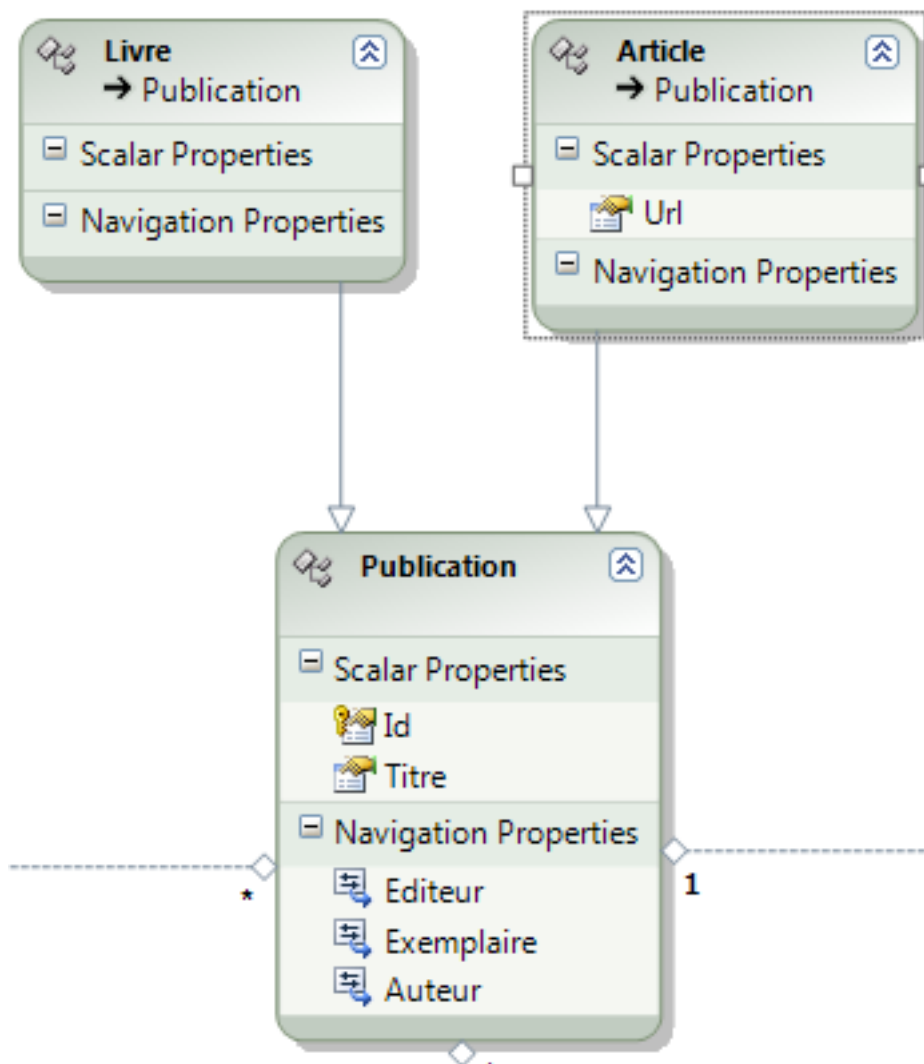
Il faut ensuite définir les propriétés de mapping. Sélectionnez la table "Publication" dans le champ "Add a Table or View". Ajoutez une condition sur la colonne "Type", laissez l'opérateur à égal et définissez la valeur à 0.



Créez ensuite l'entité "Article" de la même manière que "Livre". Ajoutez cependant une propriété à l'entité ("Add > Scalar Property") s'appelant "Url" et de type "String". Dans les propriétés de mapping, choisissez la table "Publication" et définissez une condition sur la colonne "Type" de valeur égale à 1. Au niveau de la colonne "Url", choisissez la propriété Url de type "String".



Voici le résultat de l'héritage en image :



2.2 - Les procédures stockées

Entity Framework supporte les procédures stockées. Pour rappel, les procédures stockées sont des fonctions ou méthodes stockées et exécutées par le moteur de bases de données.

Elles permettent d'effectuer les 4 types d'opérations fondamentales sur une base de données qui sont :

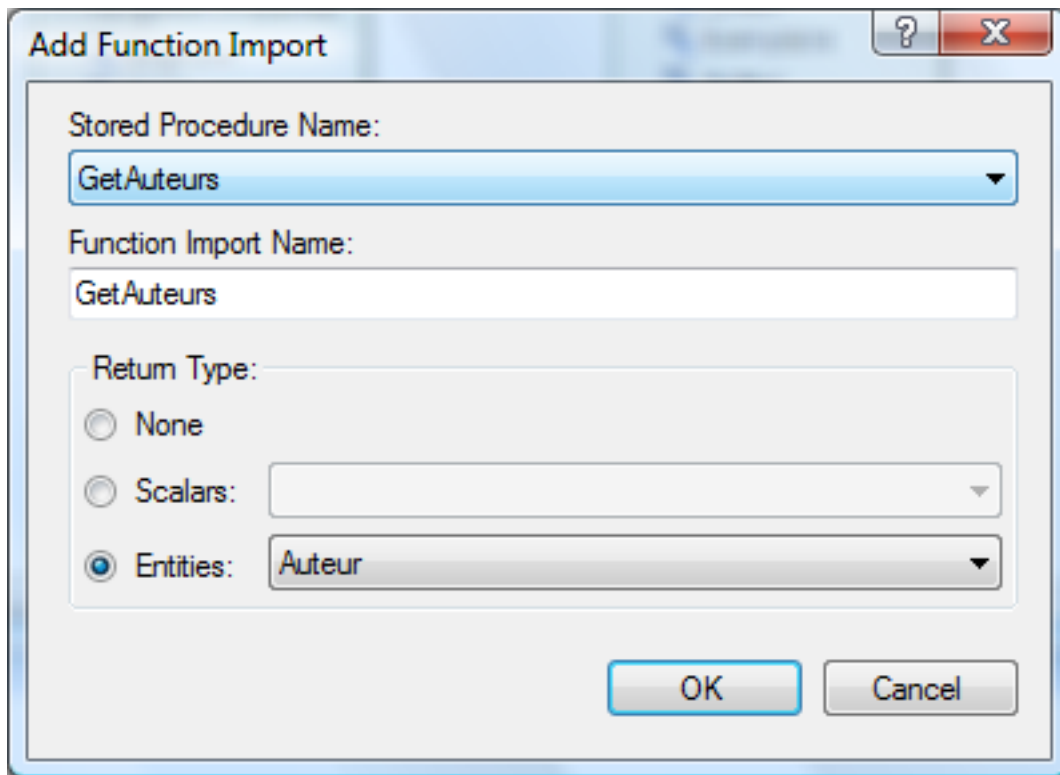
- Insertion de données (INSERT)
- Lecture de données (SELECT)
- Mise à jour de données (UPDATE)
- Suppression de données (DELETE)

Elles proposent aussi des avantages convaincants concernant la sécurité, les performances et l'encapsulation. Ce dernier point souligne le fait qu'une procédure stockée peut renfermer une logique complexe concernant, par exemple, de la mise à jour de données. Cela évite à chaque programme différent de devoir reprogrammer cette logique. Un simple appel à la procédure stockée suffit.

2.2.1 - Import de fonctions

Dans Entity Framework, une procédure stockée doit être importée pour pouvoir être appelée. Celle-ci peut être appelée via l'objet contexte.

Voyons comment définir et exécuter une procédure stockée avec Entity Framework. Dans l'éditeur de modèle, affichez l'explorateur de modèle (s'il n'est pas affiché, cliquez sur "View > Other Window > Model Browser"). Ouvrez le nœud "HelloEntityFramework.Store", ainsi que le nœud "Stored Procedures". Cliquez droit sur la procédure "GetAuteurs" et sur l'item "Create Function Import". La fenêtre suivante s'ouvre :



Dans l'encadré "Return Type", sélectionnez "Entities" et dans la liste, sélectionnez "Auteur". La procédure stockée "GetAuteurs" renvoie l'ensemble des auteurs contenus dans la table "Auteur". Validez en appuyant sur "OK".

Cette opération rajoute une fonction nommée à l'objet contexte "Modele". Pour l'exécuter, il suffit d'appeler la méthode C# nommée "GetAuteurs" de la classe "Modele". Voici un exemple :

```
public static void AppelProcédureStockée()
{
    using (Modele bdd = new Modele())
    {
        var requete = bdd.GetAuteurs();
        requete.ToList().ForEach(a => Console.WriteLine("Auteur : ID = {0}, Nom = {1}, Prenom = {2}",
            a.Id, a.Nom, a.Prenom));
    }
}
```

2.2.2 - Création, mise à jour et suppression d'entités

Il est aussi possible de définir des procédures stockées qui sont appelées lors d'une mise à jour, d'une suppression ou d'une création d'entité. Cela évite à Entity Framework de générer le SQL standard nécessaire pour les requêtes de mise à jour, de suppression ou création d'entité.

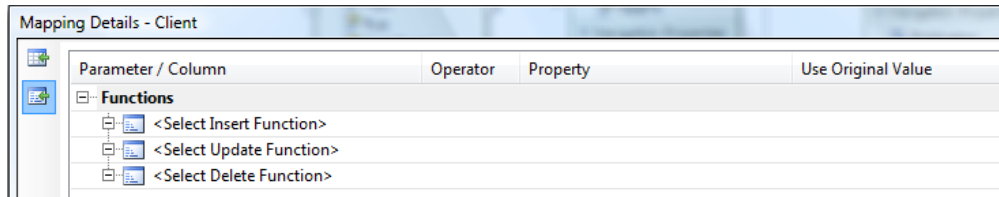
Le script SQL d'installation de la base de données "helloentityfx" a créé 3 procédures stockées servant pour cet exemple :

- "CreerClient" prend en entrée le nom, le prénom, le pays, la rue, le code postal, le téléphone ainsi que la ville du client et renvoie l'identifiant généré par la base de données.

- "MAJClient" prend en entrée l'id, le nom, le prénom, le pays, la rue, le code postal, le téléphone, la ville et met à jour le client ayant le même id que celui passé en paramètre.
- "SupprimerClient" prend en entrée l'id d'un client et supprime l'entrée correspondante dans la table.

D'une façon générale, toutes les procédures stockées servant de substitution à des requêtes de mise à jour, suppression ou création doivent avoir la même forme et la même logique que les 3 précédentes. Pour plus d'informations sur ces procédures stockées, je vous invite à regarder le script de création de la base de données "DB-helloentityfx.sql" fourni dans l'archive de l'article.

Pour utiliser ces procédures stockées, ouvrez le fichier "modele.edmx" avec l'éditeur de modèle EDM. Cliquez droit sur l'entité "Client" et sélectionnez "Stored Procedure Mapping". Cet éditeur apparaît :




Cliquez sur "<Select Insert Function>" et choisissez la procédure stockée "CreerClient". L'éditeur se charge ensuite de relier chaque propriété de l'entité aux arguments de la procédure stockée. Il ne reste plus qu'à définir le nom de la variable renvoyée par la procédure stockée contenant le nouvel identifiant et de le relier à la propriété "Id" de l'entité. Pour cela, entrez "Id" dans la case "<Add Result Binding>" juste en dessous du dossier "Result Column Bindings".

Effectuez la même manipulation pour la procédure stockée utilisée pour la mise à jour et la suppression. A ceci près qu'il n'est pas nécessaire de définir de valeur de retour pour la fonction de mise à jour. Voici le résultat en image :

Mapping Details - Client

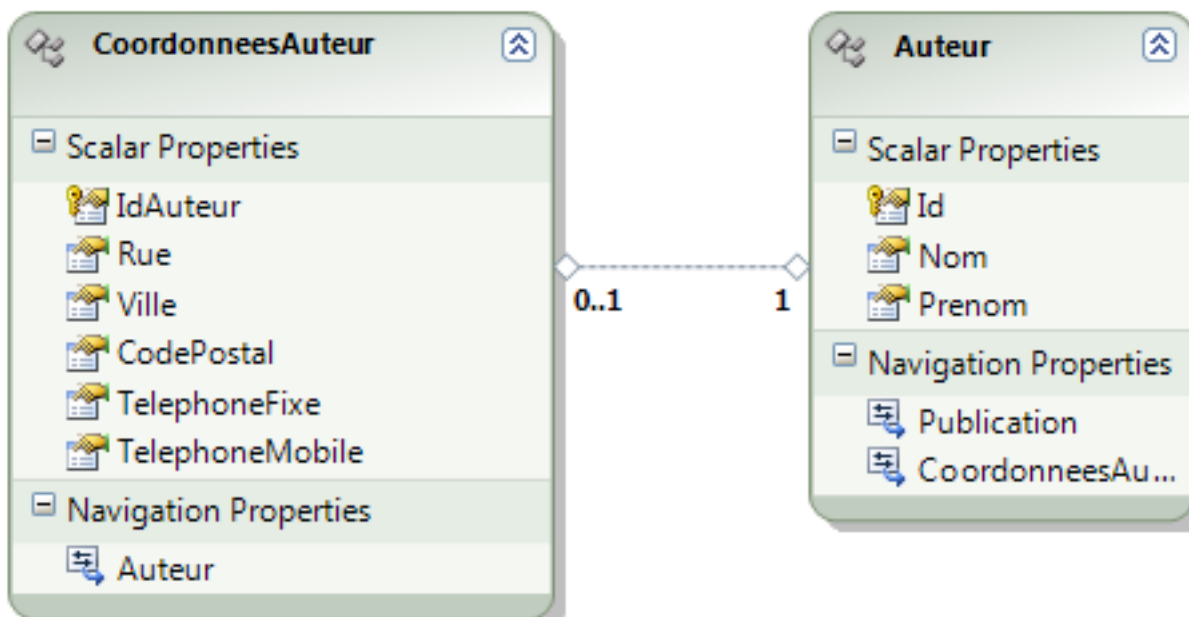
Parameter / Column	Operator	Property	Use Original Value
Functions			
Insert Using CreerClient			
Parameters			
@ Nom : varchar	←	Nom : String	
@ Prenom : varchar	←	Prenom : String	
@ Pays : varchar	←	Pays : String	
@ Rue : varchar	←	Rue : String	
@ CodePostal : char	←	CodePostal : String	
@ Telephone : char	←	Telephone : String	
@ Ville : varchar	←	Ville : String	
Result Column Bindings			
Id	→	Id : Int32	
<Add Result Binding>			
Update Using MAJClient			
Parameters			
@ Id : int	←	Id : Int32	<input type="checkbox"/>
@ Nom : varchar	←	Nom : String	<input type="checkbox"/>
@ Prenom : varchar	←	Prenom : String	<input type="checkbox"/>
@ Pays : varchar	←	Pays : String	<input type="checkbox"/>
@ Rue : varchar	←	Rue : String	<input type="checkbox"/>
@ CodePostal : char	←	CodePostal : String	<input type="checkbox"/>
@ Telephone : char	←	Telephone : String	<input type="checkbox"/>
@ Ville : varchar	←	Ville : String	<input type="checkbox"/>
Result Column Bindings			
<Add Result Binding>			
Delete Using SupprimerClient			
Parameters			
@ Id : int	←	Id : Int32	

 Il n'est pas possible de définir une seule voire 2 procédures sur les 3. Soit vous n'en définissez aucune, soit vous les définissez toutes.

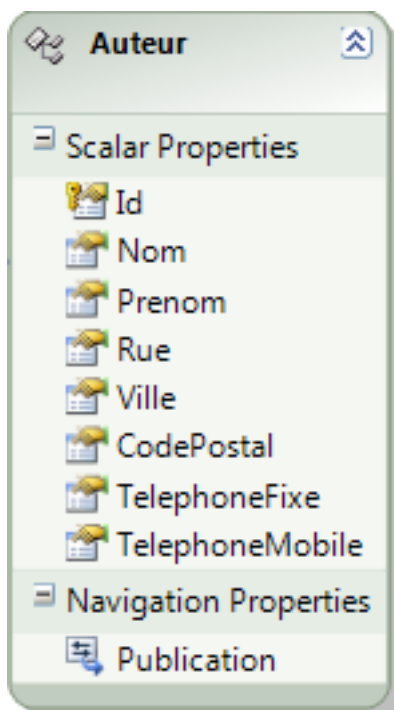
2.3 - Une entité pour 2 tables

Il est possible de mapper 1 entité sur 2 tables lorsqu'il existe entre elles une relation 0..1 -> 1.

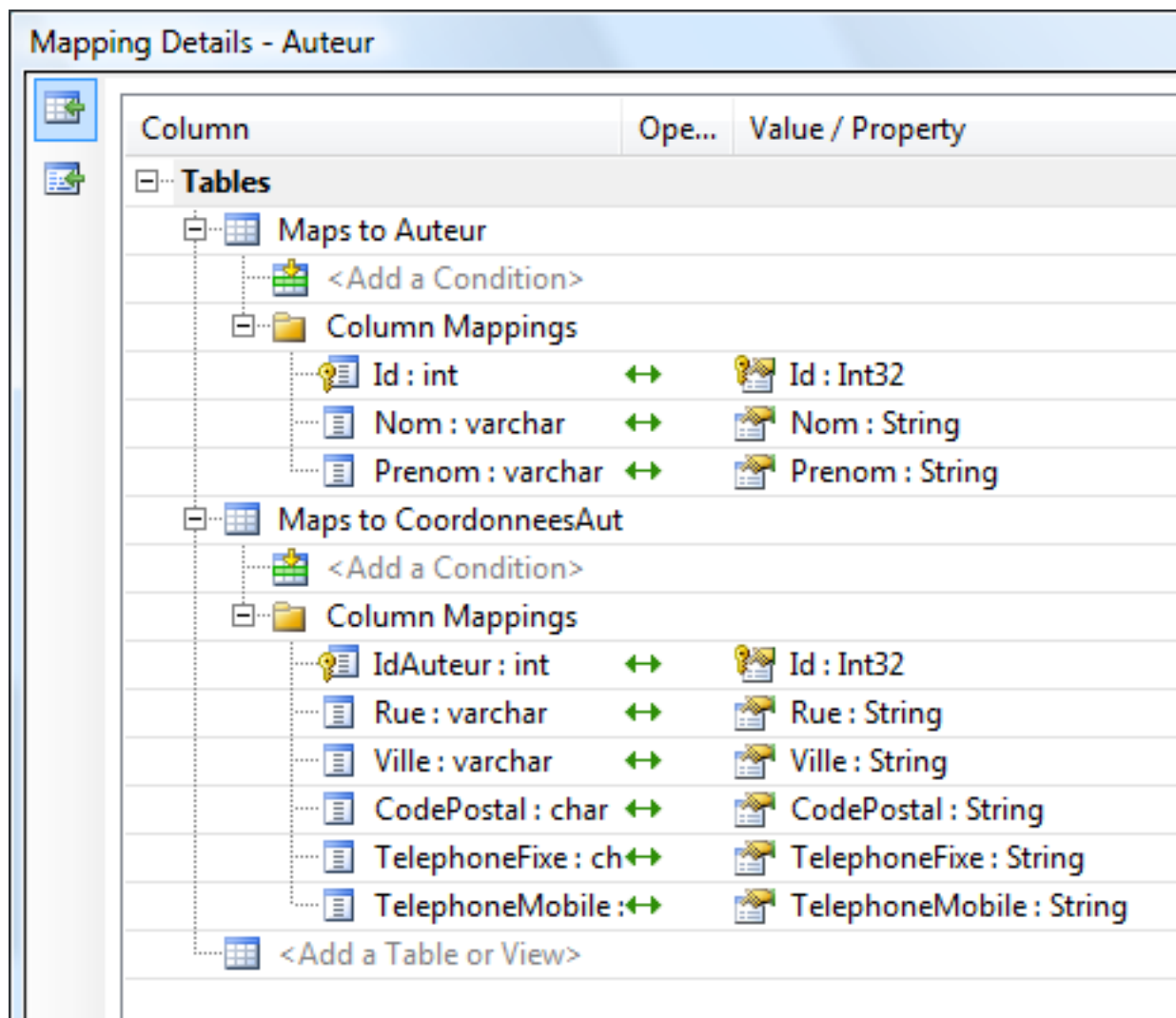
Dans le schéma "modele.edmx", il existe une entité "CoordonneesAuteur" reliée à "Auteur" par une relation 0..1 -> 1 :



Supprimez la table "CoordonneesAuteur" et ajoutez les propriétés "Rue", "Ville", "CodePostal", "TelephoneFixe" et "TelephoneMobile" à l'entité "Auteur", comme ceci :



Chaque propriété insérée doit être de type "String". Ensuite, dans les détails du mapping, il faut ajouter une relation à la table "CoordonneesAuteur". Normalement, l'éditeur assigne automatiquement tous les champs de la table aux propriétés ayant le même nom dans l'entité "Auteur", sauf pour "IdAuteur" qu'il faut spécifier avec la propriété "Id" (mappée sur le champ Id de la table "Auteur").



Sauvegardez pour déclencher le custom tool qui est chargé de générer le code, les entités et l'objet contexte. L'entité "Auteur" contient maintenant toutes les propriétés correspondantes aux champs des 2 tables "Auteur" et "CoordonneesAuteur".

Le fait de renseigner un des 5 champs ajoutés à l'entité "Auteur" crée une nouvelle entrée dans la table "CoordonneesAuteur".

2.4 - Les types complexes

Il est possible de définir des types complexes grâce à Entity Framework.

Prenons l'exemple de la table "Client". Celle-ci contient entre autres les champs suivants :

- CodePostal
- Pays
- Rue
- Telephone
- Ville

Ces champs pourraient faire référence à une nouvelle entité s'appelant "Adresse" et contenant 5 propriétés correspondant à ces champs. La classe client perdrait donc ces 5 propriétés mais aurait une propriété nommée "Adresse" de type "Adresse".

Cette entité a la particularité de ne contenir aucune clé d'entité et donc de n'être mappée à aucune clé primaire. L'entité décrite est simplement utilisée par une autre entité qui elle, contient bien une clé d'entité valide. Dans Entity Framework, cela s'appelle un type complexe.

Avec l'outil en ligne de commande "EdmGen.exe" se trouvant dans le dossier suivant "C:\Windows\Microsoft.NET\Framework\v3.5", il est tout à fait possible de définir un type complexe. Cependant, ce n'est pas possible avec l'éditeur de modèle fourni dans Visual Studio, en tout cas, pas dans sa 1ère version.

Si vous souhaitez mettre en place un type complexe, voici un [lien](#) vers la documentation MSDN.

3 - Utilisation des entités et du contexte

3.1 - Opérations CRUD

Entity Framework supporte les 4 types d'opérations désignées par l'acronyme CRUD :

- Création d'entité (Create = Création)
- Sélection d'entité (Read = Lecture)
- Suppression d'entité (Delete = Suppression)
- Mise à jour d'entité (Update = Mise à jour)

Cette partie décrit chacune des 4 opérations.

3.1.1 - Lecture des données

Entity Framework offre 3 types de possibilité pour requêter la source de données :

- LinQ To Entities
- Entity SQL
- Les méthodes génératrices de requête de "ObjectQuery<T>"

LinQ To Entities permet d'écrire des requêtes syntaxiquement proches du SQL, et cela directement en C# ou VB.Net. L'avantage majeur vient du fait que les requêtes produites sont vérifiées à la compilation et non plus à l'exécution. Comme pour chaque implémentation de LinQ, chaque requête est transformée en un arbre d'expression (créé par l'objetObjectContext) qui est au final traduit en SQL par le fournisseur de données final. Le résultat d'une requête LinQ to Entities est de type "IQueryable<T>" où T est le type spécifié dans la clause SELECT de la requête.

Voici donc un exemple de requête LinQ to Entities :

```
var requete = from publication in bdd.Publication
              select publication;
```

Le résultat est de type "IQueryable<Publication>". Il est possible d'itérer sur les résultats avec une simple boucle "foreach". Vu que "IQueryable<T>" implémente un énumérateur, sachez que la requête vers la source de données est uniquement exécutée lors du parcours des résultats. Pour plus d'information, veuillez vous référer à la documentation officielle de l'objet ["IQueryable<Publication>"](#).

LinQ to Entities supporte un ensemble de méthodes pour requêter la source de données dont voici une liste non exhaustive ([url](#) de la liste complète) :

- Select, SelectMany
- Where
- GroupJoin, Join
- All, Any
- Distinct
- Except
- Intersect
- Union
- OrderBy, OrderByDescending, ThenBy, ThenByDescending
- GroupBy
- Average
- Count, LongCount
- Max, Min, Sum

- OfType, Cast
- First, FirstOrDefault
- Skip, Take

Voici l'exemple d'une requête sélectionnant seulement les publications qui contiennent le mot "C#" dans leur titre :

```
var requete = from publication in bdd.Publication
              where publication.Titre.Contains("C#")
              select publication;
```

Pour plus d'informations sur les requêtes LinQ, je vous invite à lire l'[article de Thomas Lebrun](#).

Il est aussi possible de requêter la source de données grâce à Entity SQL. C'est un langage héritant du vocabulaire et de la grammaire de type SQL. Ce langage permet de récupérer des produits scalaires, des ensembles de résultat et des graphes d'objets. Pour une description du langage, veuillez vous référer à la [documentation officielle](#).

Voici l'exemple d'une requête récupérant toutes les publications stockées en base :

```
string chaineRequete = "SELECT VALUE p FROM Publication AS p";

// Définition de la requête de type ObjectQuery<Publication>
ObjectQuery<Publication> requete = new ObjectQuery<Publication>(chaineRequete, bdd,
    MergeOption.NoTracking);
```

En dernier choix, vous pouvez utiliser les méthodes du générateur de requêtes de la classe "ObjectQuery<T>". En voici un exemple :

```
// Définition de la requête de type ObjectQuery<Publication>
var requete = bdd.Publication.SelectValue<Publication>("it");
```

La suite de l'article fera uniquement référence à LinQ to Entities, mais toutes les requêtes faites sur le modèle peuvent être réalisées avec Entity SQL ou grâce aux méthodes du générateur de requêtes de la classe "ObjectQuery<T>".

Chaque entité en relation avec une autre possède une collection d'entités connexes. Cependant, celle-ci n'est pas initialisée par défaut. Pour naviguer vers ces entités, il est nécessaire de les charger explicitement :

- soit en utilisant la méthode "Include" de "ObjectQuery<T>",
- soit en utilisant la méthode "Load" appartenant à la classe "EntityReference<T>" ou à "EntityCollection<T>".

L'exemple suivant utilise la méthode "Include" pour récupérer explicitement, lors de la requête, l'éditeur et les auteurs associés aux publications :

```
using (Modele bdd = new Modele())
{
    var requete = from publication in bdd.Publication.Include("Auteur").Include("Editeur") select
        publication;

    requete.ToList().ForEach((Publication p) =>
    {
        Console.WriteLine("Publication : ID = {0} | Titre = {1}", p.Id, p.Titre);

        Console.WriteLine("    Editeur : ID = {0} | Nom = {1}", p.Editeur.Id, p.Editeur.Nom);

        p.Auteur.ToList().ForEach(a =>
        Console.WriteLine("    Auteur : ID = {0} | Nom = {1} {2}", a.Id, a.Prenom, a.Nom));
    });
}
```

Voici un autre exemple avec l'utilisation de la méthode "Load". Remarquez que chaque collection ou chaque référence liée à une entité possède une propriété "IsLoaded" permettant de savoir si la ou les entités connexes sont chargées.

```
using (Modele bdd = new Modele())
{
    var requete = from publication in bdd.Publication select publication;

    requete.ToList().ForEach((Publication p) =>
    {
        if (!p.EditeurReference.IsLoaded)
            p.EditeurReference.Load();

        Console.WriteLine("    Editeur : ID = {0} | Nom = {1}", p.Editeur.Id, p.Editeur.Nom);

        if (!p.Auteur.IsLoaded)
            p.Auteur.Load();

        p.Auteur.ToList().ForEach(a => Console.WriteLine("    Auteur : ID = {0} | Nom = {1} {2}",
            a.Id, a.Prenom, a.Nom));
    });
}
```

3.1.2 - Création, mise à jour et suppression

Ajouter une entité se fait très simplement en créant l'entité souhaitée et en l'ajoutant au contexte via une de ses méthodes spécialisées. En fait, le custom tool chargé de la génération du contexte et des entités, génère pour chaque ensemble (entity set) une méthode d'ajout d'entité dans le contexte. Voici un exemple créant un nouvel éditeur informatique et l'ajoutant au contexte via la méthode "AddToEditeur" :

```
using (Modele bdd = new Modele())
{
    EditeurInformatique OReilly = new EditeurInformatique()
    {
        Nom = "OReilly",
        NombreLivresInfoEdites = 1800
    };

    bdd.AddToEditeur(OReilly);
    bdd.SaveChanges();
}
```

Pour supprimer une entité, il suffit d'indiquer au contexte via la méthode "DeleteObject" l'entité souhaitée, et de valider l'action avec la méthode "SaveChanges" :

```
using (Modele bdd = new Modele())
{
    var requete = from c in bdd.Client
                  where c.Nom == "Oliver"
                  select c;

    var client = requete.FirstOrDefault();

    if (client != null)
    {
        bdd.DeleteObject(client);
        bdd.SaveChanges();
    }
}
```

Concernant la mise à jour d'entités, modifiez simplement les propriétés souhaitées et validez avec la méthode "SaveChanges". Grâce à l'objet de type "ObjectStateManager", le contexte sait quelles propriétés doivent être mises à jour lors de la mise à jour de la source de données, à la condition que l'entité soit bien attachée au contexte.

```
using (Modele bdd = new Modele())
{
    Client client = (from c in bdd.Client
                    where c.Nom == "Oliver"
                    select c).FirstOrDefault();

    if (client != null)
    {
        client.Ville = "Bruxelles";
        client.Pays = "Belgique";
        bdd.SaveChanges();
    }
}
```

Mettre à jour une relation entre 2 entités est tout aussi simple, il suffit d'assigner à la propriété adéquate l'entité connexe, tout du moins pour une relation * -> 1. Pour une relation de type * -> *, il faut ajouter la ou les entités connexes avec la méthode "Add" de la propriété faisant référence à la relation.

```
using (Modele bdd = new Modele())
{
    Emprunt emprunt = new Emprunt()
    {
        Client = bdd.Client.First(),
        DateDebut = DateTime.Now,
        Exemplaيرة = bdd.Exemplaيرة.First()
    };
    bdd.AddToEmprunt(emprunt);
    bdd.SaveChanges();
}
```

3.2 - Personnalisation

3.2.1 - Les classes entités

Comme chaque classe dérivant d'"EntityObject" est définie avec le mot clé "partial", il est possible d'étendre la définition de la classe pour définir de nouvelles méthodes, propriétés, variables, attributs, etc ...

Pour cela, il suffit simplement de créer une classe ayant le même espace de nom (namespace), le même nom, la même portée et implémentant le mot clé "partial". Voici un exemple qui surcharge la méthode "ToString" de la classe "Client" et implémente aussi 2 méthodes abstraites appelées lors de l'affectation d'une valeur à une propriété de la classe, (ici c'est la propriété "Prenom") :

```
namespace HelloEntityFramework
{
    public partial class Client
    {
        partial void OnPrenomChanging(string value)
        {
            Console.WriteLine("La propriété Prenom(valeur : {0}) de {1} va contenir la valeur {2}",
                Prenom, this.ToString(), value);
        }

        partial void OnPrenomChanged()
        {
            Console.WriteLine("La propriété Prenom(valeur : {0}) de {1} a été changée",
                Prenom, this.ToString());
        }

        public override string ToString()
        {
            StringBuilder strBuilder = new StringBuilder();
            strBuilder.Append("Client : ");
            strBuilder.Append("Nom = ");
        }
    }
}
```

```

        strBuilder.Append(Nom);
        strBuilder.Append("Prenom = ");
        strBuilder.Append(Prenom);
        return strBuilder.ToString();
    }
}

```

Le générateur de classes génère pour chaque propriété d'une entité 2 appels synchrones à des méthodes partielles. Celle contenant "Changing" est appelée avant l'affectation de la valeur à la propriété. Celle contenant "Changed" est appelée après l'affectation de la valeur à la propriété. Ces 2 types de méthodes offrent une bonne opportunité d'ajouter de la logique métier.

Il est à noter que si une méthode partielle n'est pas définie, celle-ci ne figure pas dans le code MSIL (langage intermédiaire généré par le compilateur C#). On peut penser à une sorte d'optimisation par rapport aux événements ordinaires.

3.2.2 - La classe de contexte

De la même manière que pour les entités, il est possible d'étendre la définition des classes dérivant de "ObjectContext".

Voici un exemple qui implémente la méthode "OnContextCreated" et qui ajoute une méthode à l'évènement "SavingChanges" :

```

namespace HelloEntityFramework
{
    public partial class Modele
    {
        partial void OnContextCreated()
        {
            SavingChanges += new EventHandler(Modele_SavingChanges);
        }

        void Modele_SavingChanges(object sender, EventArgs e)
        {
            var entitésAjoutees =
                ((ObjectContext)sender).StateManager.GetObjectStateEntries(EntityState.Added);

            foreach (ObjectStateEntry entry in entitésAjoutees)
            {
                if (!entry.IsRelationship)
                    Console.WriteLine(String.Format("Ajout d'une entité de type {0}",
                        entry.Entity.GetType().ToString()));
            }
        }
    }
}

```

La méthode "Modele_SavingChanges" reçoit en paramètre le contexte sous la forme d'un objet de type "object". Celle présentée ci-dessus affiche le type des entités ajoutées à la base de données. C'est aussi un bon emplacement pour y définir des règles ou des contraintes qui doivent être respectées.

3.3 - Sérialisation

Les classes dérivant de la classe "Entity" peuvent être sérialisées de 3 manières différentes:

- En binaire avec l'aide de la classe "BinaryFormatter". Vous trouverez  [ici](#) un exemple sur MSDN
- En XML à l'aide de la classe "XmlSerializer"

- Par WCF puisque chaque classe "Entity" implémente l'attribut "DataContract" et chaque membre l'attribut "DataMember".

Voici un exemple de sérialisation XML :

```
public static void SerialisationXML()
{
    using (Modele bdd = new Modele())
    {
        var client = bdd.Client.First();

        XmlSerializer serialiseur = new XmlSerializer(typeof(Client));
        TextWriter textWriter = new StreamWriter("client.xml");
        serialiseur.Serialize(textWriter, client);
        textWriter.Close();
    }
}
```

```
public static void DeserialisationXML()
{
    Console.WriteLine("Désérialisation d'un client contenu dans un fichier XML");

    XmlSerializer serialiseur = new XmlSerializer(typeof(Client));

    TextReader textReader = new StreamReader("client.xml");
    Client client = (Client)serialiseur.Deserialize(textReader);
    textReader.Close();

    Console.WriteLine(client.ToString());
}
```

La sérialisation WCF ainsi que binaire sérialise toutes les entités chargées et rattachées à l'entité que l'on veut sérialiser. Par contre, la sérialisation XML ne supporte pas cette fonctionnalité. Pour contourner ce comportement, il est nécessaire de rajouter de la logique permettant de rattacher les entités en relation avec l'entité voulue, et bien sérialiser une à une chaque entité.

3.4 - Relations avec le contexte


Toute entité récupérée à partir d'un objet de type "ObjectContext" est par défaut attachée au contexte. Pour des raisons de performance, il peut être utile de détacher certaines entités du contexte. Lorsque ceci est fait, le contexte ne se soucie plus de l'entité détachée et ne trace plus ses modifications. Le fait de détacher une entité ne détache pas pour autant les entités qui sont en relation avec elle. Voici un exemple de code permettant de détacher une entité :

```
public void DétacherEntité(Client pClient)
{
    using (Modele bdd = new Modele())
    {
        if (pClient != null)
            bdd.Detach(client);
    }
}
```

Il est possible d'attacher une entité qui ne l'est pas. Pour cela, la méthode "Attach" de l'objet "ObjectContext" prend en paramètre l'entité à attacher. Cependant, celle-ci est attachée avec son état défini à "Unchanged". Cela veut dire que toute modification appliquée avant l'appel à la méthode "Attach" est perdue. De même, l'entité doit avoir une clé ("EntityKey") valide. Dans le cas contraire, la méthode "AttachTo" est nécessaire.

```
public void AttacherEntité(Client pClient)
{
```

```
using (Modele bdd = new Modele())
{
    // Attacher un objet au contexte
    if (pClient != null)
        bdd.Attach(pClient);
}
}
```

 Le fait d'attacher une entité, attache aussi les entités qu'elle peut avoir en relation.

Dans le cas où une entité détachée a subi des modifications qui doivent être sauvegardées en base, vous pouvez utiliser la méthode "ApplyPropertyChanges". Celle-ci prend en paramètre le nom de l'ensemble d'entités ("Entity Set") et de l'entité contenant les modifications à sauvegarder.

```
public void MiseAJourEntité(Client pClient)
{
    using (Modele bdd = new Modele())
    {
        // Extraction et création de la clé de l'entité
        EntityKey key = bdd.CreateEntityKey("Client", pClient);

        // Essaie de récupérer l'entité originale à partir de sa clé
        object clientOriginal;
        if (bdd.TryGetObjectByKey(key, out clientOriginal))
        {
            // Applique les modifications de l'entité détaché comportant des modifications
            bdd.ApplyPropertyChanges(key.EntitySetName, pClient);
        }

        bdd.SaveChanges();
    }
}
```

3.5 - Accès concurrentiel

Entity Framework implémente un accès concurrentiel optimiste, c'est-à-dire que les données ne sont pas verrouillées lorsqu'elles sont utilisées.

Dans le cas où les données ont changé entre le cache d'Entity Framework et la source de données, il est possible de résoudre ce conflit via 2 solutions :

- Les données de la base font office de référentiel.
- Les données dans le cache d'Entity Framework font office de référentiel.

Cependant, cet accès concurrentiel n'est pas activé par défaut. Tout d'abord, l'accès concurrentiel concerne une propriété d'une entité. Si vous souhaitez l'activer sur toutes les entités et sur toutes leurs propriétés, il faut répéter la manipulation suivante autant de fois que nécessaire. Ouvrez votre modèle avec le designer de modèle EDM, sélectionnez une entité quelconque (Client par exemple), sélectionnez une propriété et dans les propriétés de celle-ci se trouve la valeur "Concurrency Mode" définie à "None" (Prénom pour l'exemple ci-dessous). Définissez là à "Fixed" pour activer la concurrence.

Une fois activée, la logique de mise à jour ou de suppression peut générer des exceptions de type "OptimisticConcurrencyException", lorsque les données du cache d'Entity Framework et de la base de données ne sont plus les mêmes au moment de l'appel à la méthode "SaveChanges" du contexte. Ce cas de figure arrive fréquemment lorsque plusieurs clients consomment et mettent à jour la même base de données. Comme cité plus haut, soit nous utilisons comme référentiel les données locales ou bien celles de la base de données. Ceci est décidé grâce à la méthode "Refresh" de l'objet contexte, qui prend en paramètre une valeur de l'énumération RefreshMode ("ClientWins" ou "StoreWins") et l'entité à rafraîchir. Voici un exemple que vous pouvez reproduire avec la base "helloentityfx":

```

public static void RefreshEntité()
{
    Console.WriteLine("Récupérer une entité en cas de conflit (Accès concurrentiel)");

    using (Modele bdd = new Modele())
    {
        Client client = null;
        try
        {
            client = (from c in bdd.Client
                    where c.Nom == "Oliver"
                    select c).FirstOrDefault();

            simulerAccesConcurrentiel(bdd.Connection.DataSource);

            client.Prenom = "Fabrice";
            bdd.SaveChanges();

            Console.WriteLine("Aucun conflit");
        }
        catch (OptimisticConcurrencyException exp)
        {
            Console.WriteLine("Conflit détecté. Message : {0}", exp.Message);

            bdd.Refresh(RefreshMode.StoreWins, client);
            bdd.SaveChanges();
        }
    }
}

private static void simulerAccesConcurrentiel(string pDataSource)
{
    using (SqlConnection connexion = new SqlConnection("Data Source=" + pDataSource +
";Initial Catalog=helloentityfx;Integrated Security=SSPI;"))
    {
        connexion.Open();

        string strCmd = "UPDATE Client SET Prenom = 'Johnny' WHERE Nom = 'Oliver'";
        SqlCommand command = new SqlCommand(strCmd, connexion);
        command.ExecuteNonQuery();

        connexion.Close();
    }
}

```

La méthode "simulerAccesConcurrentiel" utilise une connexion ADO.Net classique pour mettre à jour l'entrée de la table "Client" correspondant à l'entité précédemment récupérée, simulant un accès concurrentiel. A l'appel de la méthode "SaveChanges" dans le block "try", une exception de type "OptimisticConcurrencyException" est récupérée. Elle est ensuite affichée dans le block "catch", où nous décidons de mettre à jour l'entité en utilisant comme référentiel la base de données :

```

...
bdd.Refresh(RefreshMode.StoreWins, client);
...

```

3.6 - Transactions

Entity Framework supporte les transactions de manière automatique, tout du moins en ce qui concerne la source de données.

Les transactions sont utiles lorsqu'il est nécessaire d'exécuter plusieurs opérations comme si ce n'était qu'une seule. Si une opération de la transaction échoue en générant une exception, toutes les opérations précédentes sont annulées. D'une façon générale, une transaction respecte les 4 contraintes suivantes (ACID) :

- **Atomicité** : Une transaction doit s'exécuter totalement ou pas du tout.
- **Cohérence** : La cohérence des données doit être absolument respectée même en cas d'erreur. Il doit être possible d'annuler les opérations en cas d'erreur.
- **Isolation** : La transaction doit s'effectuer dans un mode isolé où elle seule peut voir les modifications en cours.
- **Durabilité** : Une fois la transaction terminée, le système est dans un état stable, quelque soit ce qui s'est passé durant la transaction.

Entity Framework crée une transaction lors de l'appel à la méthode "SaveChanges" du contexte. Ceci permet de maintenir un état cohérent entre la source de données et le contexte.

Si vous souhaitez ajouter d'autres opérations concernant des systèmes externes, comme un annuaire, une autre base de données ou un envoi de mail, il est nécessaire de déclarer explicitement une transaction. Dans ce cas, Entity Framework utilisera automatiquement celle-ci. Une transaction distribuée a aussi besoin d'un coordinateur de transaction distribuée (DTC) et tous les systèmes externes appelés doivent être compatibles.

Pour pouvoir relancer une transaction ayant échoué, il faut appeler la méthode "SaveChanges" avec le paramètre "acceptChangesDuringSave" égal à "false", et ne pas oublier l'appel à la méthode "AcceptAllChanges" une fois la transaction validée. Voici un exemple avec le modèle de l'article :

```
int nbEssai = 3;
bool transactionExecutee = false;

using (Modele bdd = new Modele())
{
    while (nbEssai > 0 && !transactionExecutee)
    {
        try
        {
            Client client = new Client()
            {
                Nom = "Prenom",
                Prenom = "Paul"
            };

            bdd.AddToClient(client);
            bdd.SaveChanges(false);
            transactionExecutee = true;
        }
        catch (UpdateException exp)
        {
            Console.WriteLine("Erreur détectée : {0}", exp.Message);
            nbEssai--;
        }
    }
    if ( transactionExecutee )
        bdd.AcceptAllChanges();
}
```

Conclusion

Entity Framework représente la solution tant attendue de mapping objet-relationnel de Microsoft. Elle constitue une bonne alternative à des outils comme NHibernate, grâce à son architecture ingénieuse du modèle EDM et des entités. L'éditeur de modèle EDM de Visual Studio 2008 permet aussi de faire gagner un temps précieux dans la définition des entités et de leur mapping. L'équipe d'ADO .Net prépare en ce moment une version 2 corrigeant certaines lacunes concernant l'utilisation des classes POCO et du designer.

Remerciements

Je tiens à remercier Jérôme Lambert, pvalatte et Louis-Guillaume Morand pour leurs conseils et encouragements ainsi que Maximilian pour la correction orthographique.