ADO.NET (version C#)

J-M Rabilloud – S Curutchet

Remerciements

Nous adressons ici tous nos remerciements à l'équipe de rédaction de "developpez.com" et tout particulièrement à Etienne Bar, Emilie Guittier, Maxence Hubiche et David Pédehourcq pour le temps qu'ils ont bien voulu passer à la correction et à l'amélioration de cet article.

Introduction

Avec la nouvelle plate-forme DotNet, Microsoft® fournit une nouvelle architecture d'accès aux données appelée ADO.NET. Comme à notre habitude, nous allons visiter en détail les objets la composant avec des exemples d'utilisation. Cependant, dans la première partie, nous allons voir ce qu'il en est pour le développeur ADO. Ce cours risque d'être assez dense. La bonne compréhension de celui-ci suppose que vous ayez assimilé correctement les concepts fondamentaux d'ADO. Un certain nombre d'autres connaissances vous seront nécessaires, même si nous nous livrerons à quelques séances de rappel dans ce cours. Developpez.com met à votre disposition de nombreux documents vous permettant de vous aider.

ADO.NET

Les objets Connection, Command et Datareader dans **ADO.NET**
Les objets DataAdapter et DataSet dans ADO.NET par Leduke

SQL

Le site de référence de Frédéric Brouard

ADO

Utiliser les Recordset ADO
VB-VBA : **ADOX** structure & sécurité des données par moi-même

XML

Le **Cours** en Français sur **XML-XLS** de Luc Van Lancker
Le **document** de référence des spécifications **XML**

Cette version de ce cours utiliseras le langage C# sur le Microsoft® .NET FrameWork 1.1.

La plupart des exemples de ce cours vont utiliser les bases de données Pubs pour SQL-SERVER et Biblio pour Access 2000.

Bonne lecture

Passer d'ADO à ADO.NET

Les limitations du modèle ADO

Tous les développeurs ayant eu à utiliser ADO en mode client et a fortiori en mode déconnecté se sont heurtés aux limitations induites par le moteur de curseur client ADO. Il n'est pas possible de travailler, sauf en utilisant une programmation extrêmement lourde, sur un modèle relationnel de données. La seule technique acceptable consistait à multiplier le nombre d'interventions directes sur la source de données ce qui avait deux effets pervers :

Une certaine 'déconnexion' entre le Recordset et la source. En effet, l'action par le biais de commande indépendante obligeait la synchronisation fréquente du Recordset. Ceci impliquait une sollicitation importante de la connexion, ce qui est le but inverse de la programmation côté client

La perte quasi-totale de la programmation générique. L'écriture de commande SQL devait tenir compte des spécificités SQL du fournisseur ce qui impliquait un code dédié pour le fournisseur et la perte d'un des plus gros avantages d'ADO.

D'autres limitations, dues à l'utilisation du modèle COM, posaient aussi des problèmes de conversion de types ou de franchissement.

Dans le cadre d'un développement WEB, où les ressources du serveur sont comptées, il est indispensable d'avoir un mode de travail côté client beaucoup plus performant. Avec l'arrivée des Web Forms de la plate-forme DotNet, ADO ne pouvait pas suffire pour un travail déconnecté performant.

ADO.NET la révolution

Je prends ce titre provocateur pour tuer le mythe tout de suite. S'il y a une véritable révolution entre VB6 et VB.NET il n'en est pas du tout de même pour le passage vers ADO.NET.

Il existe de nombreux articles sur le NET traitant d'ADO et d'ADO.NET. Lors des recherches préliminaires à l'écriture de ce cours j'en ai lu plusieurs tant en français qu'en anglais, et il faut bien constater qu'une grande quantité d'idées reçues oscillant entre le mythe et l'ignorance hantent une grande partie de la prose libre traitant de ce sujet. Nous allons tenter de rétablir ici une vision saine de la problématique. Attention, je ne vois pas d'intention malveillante derrière les idées reçues véhiculées dans ces articles, seule la profonde méconnaissance d'ADO est à mettre en cause. J'en veux pour preuve les interprétations variées de ce qu'est un curseur côté serveur dans le modèle ADO.

ADO.NET remplace complètement ADO

Cette vision est fausse et qui plus est dangereuse. ADO.NET de son premier nom ADO+ est une extension d'ADO, et est annoncée comme telle par Microsoft®. Pour schématiser, il s'agit d'une révision des composants RDS et du travail sur les jeux de données déconnectés. Cette remise à plat était d'ailleurs nécessaire et l'approche ADO.NET est un indéniable progrès.

Il faut tout réapprendre pour passer d'ADO à ADO.NET

Voilà sûrement le plus gros des serpents de mer. Il est amusant d'ailleurs de constater qu'il est diversement employé selon que l'on sera attiré ou repoussé par ADO.NET. Bien que nombreuses, les variations les plus communes sont les suivantes.

Dans ADO.NET les curseurs ont disparus

Puisque ADO.NET se centre sur la gestion des jeux de données client déconnectés, il est bien certain qu'il n'y a guère d'intérêt à gérer les curseurs avec ADO.NET. La présence du DataReader dans le modèle ADO.NET sert d'ailleurs d'argument massue dans ce contexte. Pourtant le DataReader, qui n'est rien d'autre qu'un curseur serveur de type FireHose a sa raison d'être dans une approche WEB qui est celle d'ADO.NET. En effet pour les très gros jeux d'enregistrements comme pour la récupération d'une valeur unique (ExecuteScalar) il fallait bien pouvoir utiliser autre chose qu'un DataSet.

Avec DotNet impossible de travailler en mode connecté.

La c'est l'amalgame qui prime. Avec ADO.NET assurément non. Mais la plate-forme permet l'utilisation d'ADO, cette utilisation étant documentée par Microsoft®. Nous reviendrons d'ailleurs en détails sur le quand et le comment de l'emploi de cette solution.

Il faut choisir ADO ou ADO.NET

C'est tout à fait faux. Il est tout à fait possible d'employer les deux dans un même projet, il est tout autant possible de convertir un Dataset en Recordset et inversement. En fait tout est question de besoin.

La migration d'ADO vers ADO.NET est complexe

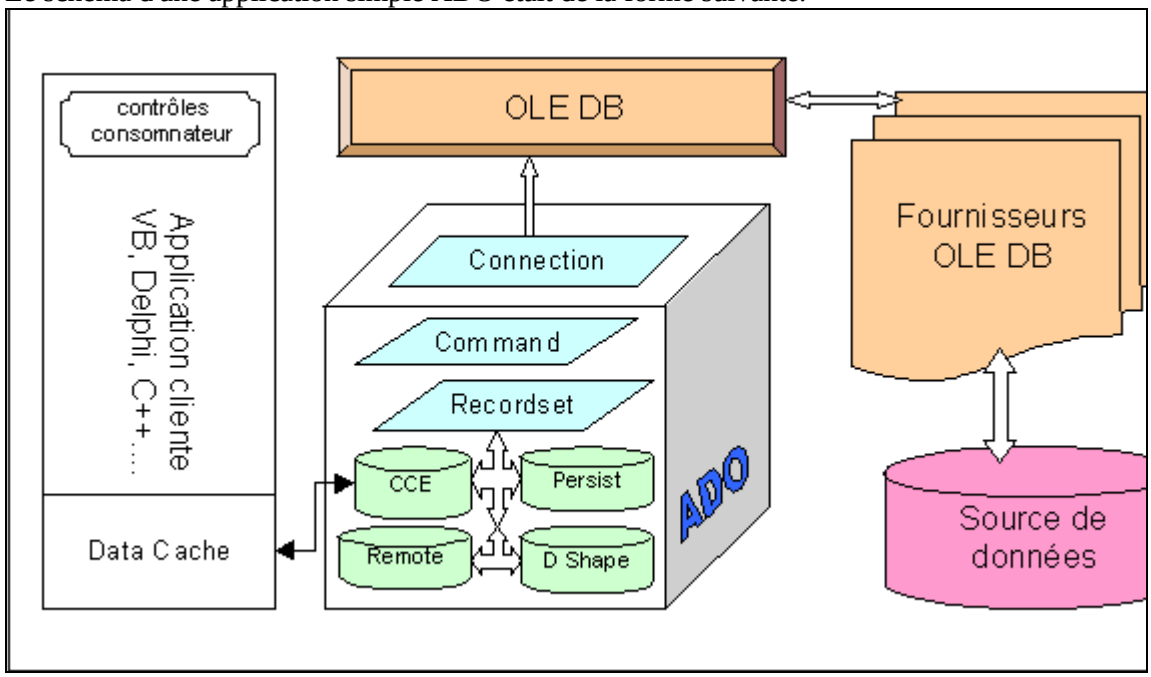
Toutes vos connaissances ADO vont continuer de vous être indispensables pour utiliser ADO.NET et le passage va se faire sans douleur. ADO.NET n'est pas un remplacement total d'ADO mais bien un enrichissement de celui-ci. Ceci ne veut pas dire qu'il n'est pas possible d'apprendre à utiliser ADO.NET sans avoir préalablement appris à utiliser ADO. Les concepts de fonctionnement sont exactement les mêmes et l'apprentissage de l'un permet un apprentissage aisé de l'autre. Bien sur, il y aura une part non négligeable de réécriture de code comme nous le verrons un peu plus loin.

Nous allons maintenant voir ce qui a changé, quels sont ces nouveaux concepts tellement effroyables qu'ils forcent le re codage de toutes vos applications, bref allons découvrir ensemble de quoi la bête est faite.

ADO.NET une évolution nécessaire

Avec ADO il existe deux méthodes pour obtenir un jeu d'enregistrements déconnecté, et deux formats pour le manipuler. Pour obtenir un jeu d'enregistrements déconnecté, on employait soit un composant RDS (Remote Data Access) dans le cas des applications Web, soit un Recordset client dont on coupait la connexion. Ces jeux d'enregistrements pouvaient être rendu persistants soit dans un format propriétaire (Datagram – ADTG) soit dans un format libre (Extensible Markup Language – XML). Ceci était certes suffisant pour écrire des applications simples en mode déconnecté mais il subsistait des problèmes conséquents. De plus, ADO est basé sur COM et en cela subit les problèmes de communications inter-process et de types.

Le schéma d'une application simple ADO était de la forme suivante.



Dans certains cas, tout se passe dans le même process, dans d'autres la source de données se trouve dans un autre process. Examinons notre boîte ADO.

Dedans, outre mes classes standards, se trouvent quatre objets un peu particuliers, les fournisseurs de service.

CCE (Client Cursor Engine – Microsoft Cursor Service for OLE DB – Moteur de curseur client)

C'est le plus connu des quatre, parfois même le seul connu. Dans une base de données il n'y a pas de notion d'ordre de ligne ou de colonne, donc pas de notion d'enregistrement en cours. ADO permet au développeur par le biais de ce fournisseur d'avoir accès à une bibliothèque minimum de curseurs ce que le fournisseur ne peut garantir. De plus il permet l'accès à des fonctionnalités supplémentaires et permet toujours l'utilisation de la mise à jour par lots.

D Shape (Microsoft Data Shaping Service for OLE DB – Mise en forme des données)

Ce fournisseur permet la gestion de jeux de données hiérarchiques. Il est forcément utilisé en conjonction avec un fournisseur de données qu'il supporte.

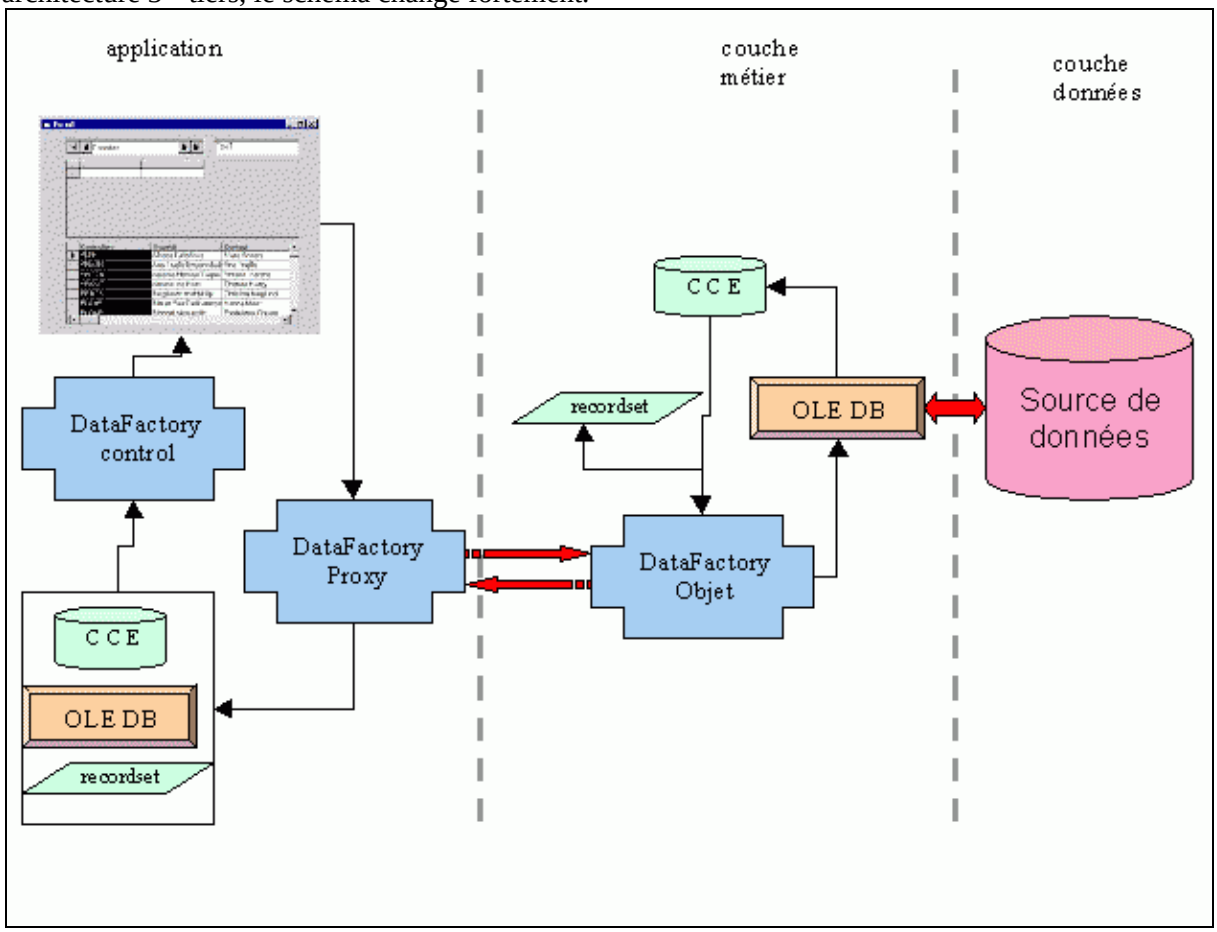
Persist (Microsoft OLE DB Persistence Provider – Fournisseur de persistance)

Ce fournisseur permet de stocker un Recordset comme un fichier. L'ensemble des données et méta-données étant sauvegardé, il est possible de le re connecter plus tard. Deux formats de fichier sont gérés, ADTG et XML.

Remote (Microsoft OLE DB Remoting Provider – Accès à distance)

Le plus méconnu des fournisseurs de service, c'est celui qui permet d'aller utiliser les fournisseurs de données d'une machine distante.

Très souvent dans le modèle au-dessus, la connexion n'est jamais fermée tant que l'application utilise des données. Il faut dire que dans le cas d'une application locale utilisant des données locales, la fermeture de la connexion ne présente que peu d'intérêt. Dans ce type de schéma on travaille d'ailleurs presque exclusivement côté serveur et c'est pour cela que l'on dit qu'ADO est un modèle fortement lié à la connexion. Cette analyse est pourtant audacieuse. Car dès lors que l'on travaille sur une source distante ou en utilisant une architecture 3+-tiers, le schéma change fortement.



Dans ce type d'architecture, on duplique fortement les objets mis en œuvre. De plus on engendre forcément des opérations de marshalling COM. Et encore, dans ce cas je n'ai rendu ni mon Recordset persistant, ni hiérarchique. Le modèle ADO est donc fortement contraignant dans ce type d'application.

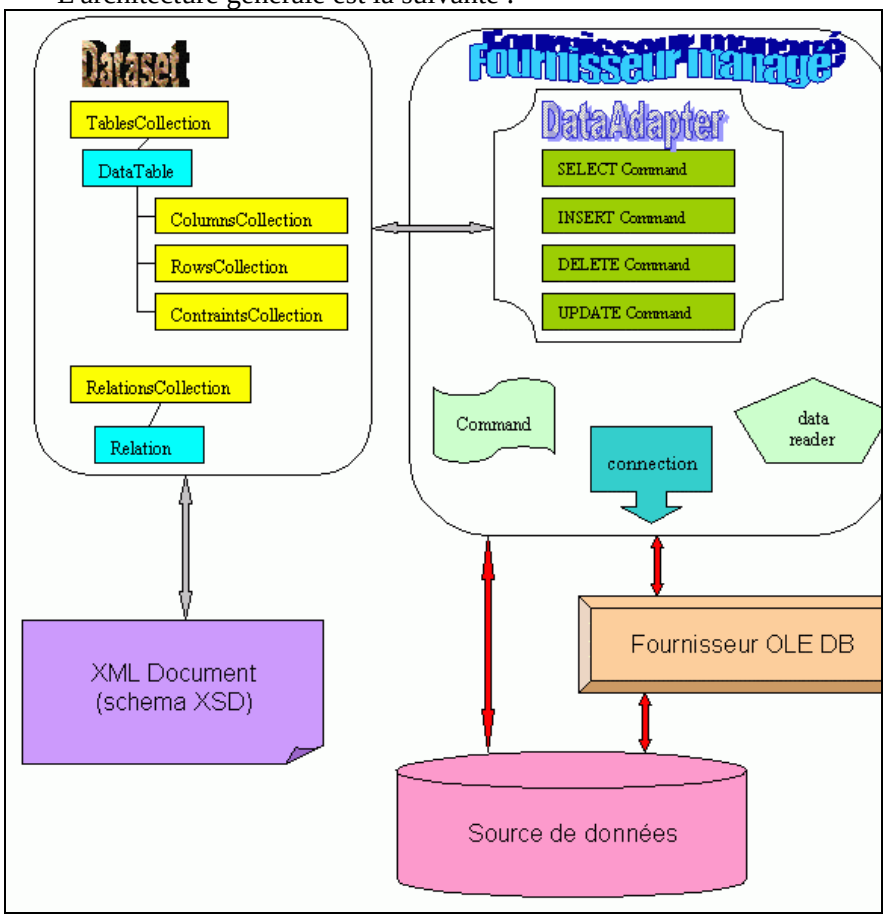
Une nouvelle architecture

Fort de cette constatation Microsoft a donc étendu son modèle avec ADO.NET.

L'architecture d'ADO.NET est, elle, très orientée vers les architectures réseaux. De plus, nous allons pouvoir nous affranchir du modèle COM. Pour cela ADO.NET utilise un nouveau concept, les fournisseurs managés. Pour schématiser on peut dire qu'il y a déconnexion du modèle. Toutes les classes pouvant communiquer avec la source de données sont regroupées dans le fournisseur managé, toutes celles liées au jeu d'enregistrements étant regroupées dans le Dataset.

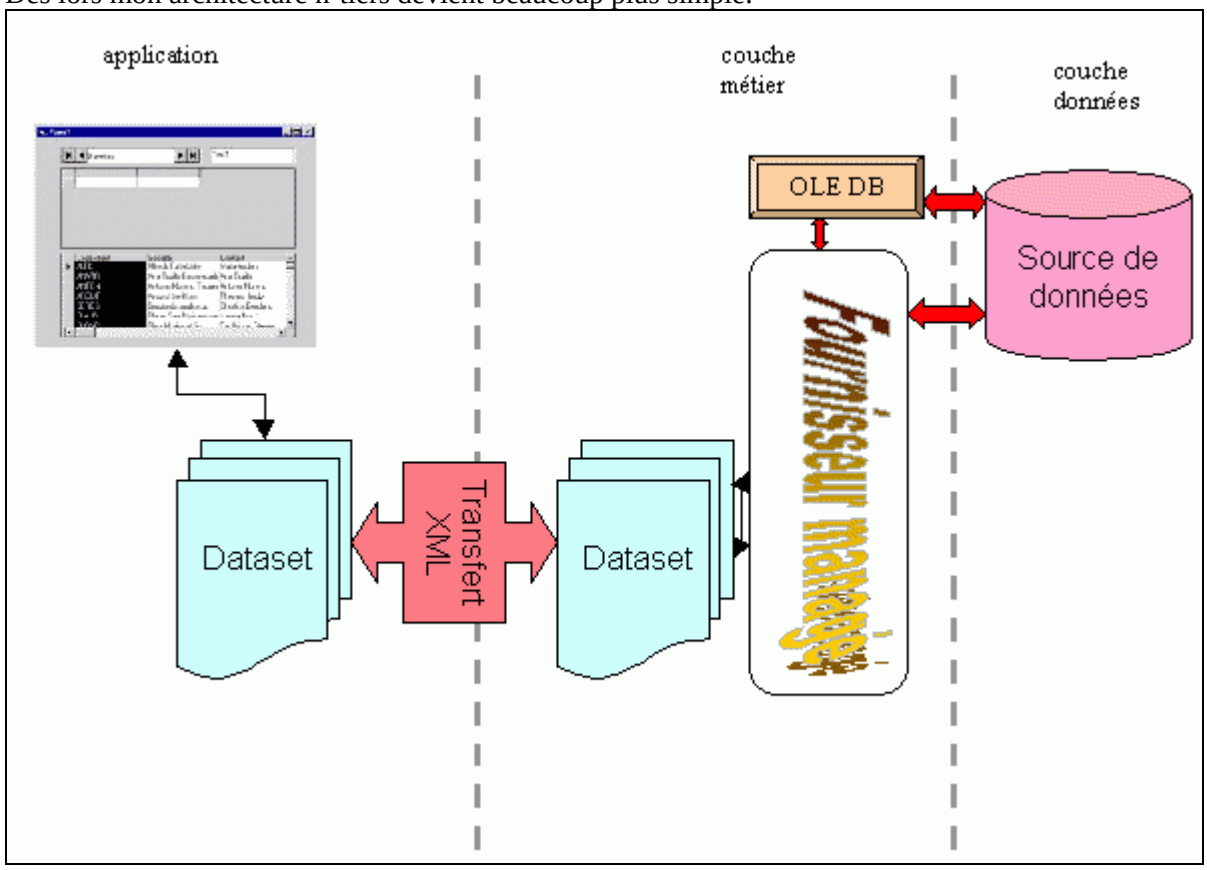
Le fournisseur managé utilise en général un fournisseur OLE DB pour communiquer avec la source de données sauf dans le cas de SQL-Server ou le protocole TDS (Tabular Data Stream) est employé.

L'architecture générale est la suivante :



Comme nous le voyons, il n'y a plus de relation directe entre le jeu d'enregistrements et la source de données. Ce terme de jeu d'enregistrements n'est d'ailleurs pas très approprié. En effet un Dataset n'est que peu comparable avec un Recordset, du fait de la déconnexion et de leurs différences de concepts, mais nous y reviendrons un peu plus loin.

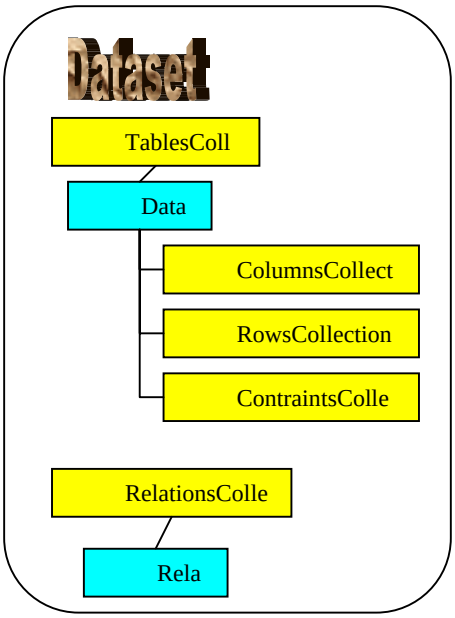
Dès lors mon architecture n-tiers devient beaucoup plus simple.



Il n'y a plus de franchissement COM à prévoir. De plus le fait qu'il s'agisse alors d'un transfert permet de franchir les Firewall ce que le modèle ADO ne pouvait pas faire aisément.

Le Dataset

C'est probablement la classe qui perturbe le plus le développeur ADO qui découvre ADO.NET. Nous allons l'examiner sur un plan assez théorique dans cette partie, nous apprendrons son fonctionnement détaillé dans la suite de l'article.



Dataset Vs Recordset

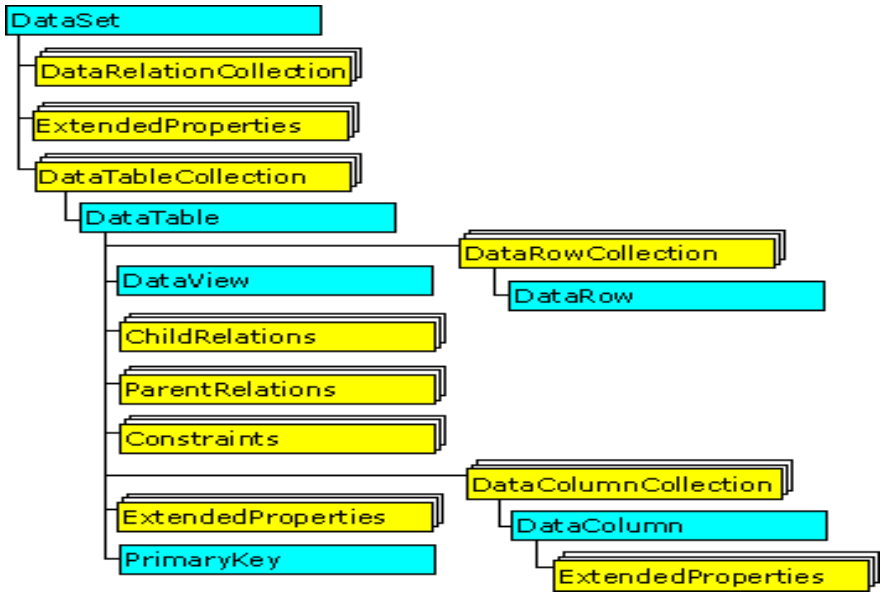
L'erreur à ne pas faire si on désire appréhender correctement le Dataset est de le comparer stricto sensu à l'objet Recordset ADO. Un Recordset ADO est sensiblement l'équivalent d'une vue SQL, c'est à dire une table produite par une requête 'SELECT' avec ou sans jointure. L'utilisation du Data Shape Provider sur ce point n'est rien d'autre qu'une présentation hiérarchisée de la même vue. Le Recordset contient aussi un certain nombre d'informations de schéma, plus ou moins juste selon la nature du curseur. Toutes les contraintes existantes dans le Recordset, ainsi que les index et les clés existent dans la source de données. Le jeu d'enregistrements est enfin piloté par un service de moteur de curseur qui régit :

- La notion d'enregistrement en cours → Dans une source de données cette notion n'existe jamais. Dans ADO elle est différente selon la position du curseur. Dans le cadre de la lecture d'un enregistrement, la position du curseur n'intervient pas, mais dès lors qu'on souhaite modifier les données, le fonctionnement diffère. Dans le cas des curseurs clients qui nous intéresse ici, cet enregistrement ne peut pas être retrouvé dans la source de données s'il n'est pas unique.
- Le déplacement dans le jeu → Dans ADO certains curseurs ont un mode de défilement obligatoire vers l'avant, les autres étant bidirectionnels. Pour ceux-ci, il est alors possible de connaître le nombre d'enregistrements rapatriés par la requête ainsi que la position de l'enregistrement en cours dans le jeu.
- La sensibilité aux modifications de la source → Les curseurs non statiques peuvent, dans une certaine mesure, refléter dynamiquement les modifications de la source de données du fait d'autres 'utilisateurs'.

Avec les Dataset, rien de tout cela. Il faut voir le Dataset comme une base de données relationnelle, locale à votre application. Celle-ci peut être la copie conforme ou tronquée d'une base existante, la résultante de données provenant de multiples sources, voire entièrement créée dans votre application. Un Dataset se compose, tout comme une base de données, de tables et de relations. Ces tables sont similaires à des tables de SGBD avec la gestion de lignes (les enregistrements), de colonnes (les champs) et de contraintes (la limitation des opérations utilisables). Les relations permettent de lier les tables entre-elles. Du fait de la déconnexion totale du Dataset, vous pouvez parfaitement modifier tous ces objets même lorsqu'ils sont issus d'une base existante.

Un Dataset n'est donc pas un jeu d'enregistrements mais un ensemble de collections représentant une base de données, stocké dans un fichier XML. En cela, la notion de déplacement ou d'enregistrement en cours doit être vu différemment, la sensibilité aux données n'existant plus du fait de la déconnexion.

Donc notre Dataset est un conteneur de collection comme nous pouvons le voir dans le schéma ci-dessous :

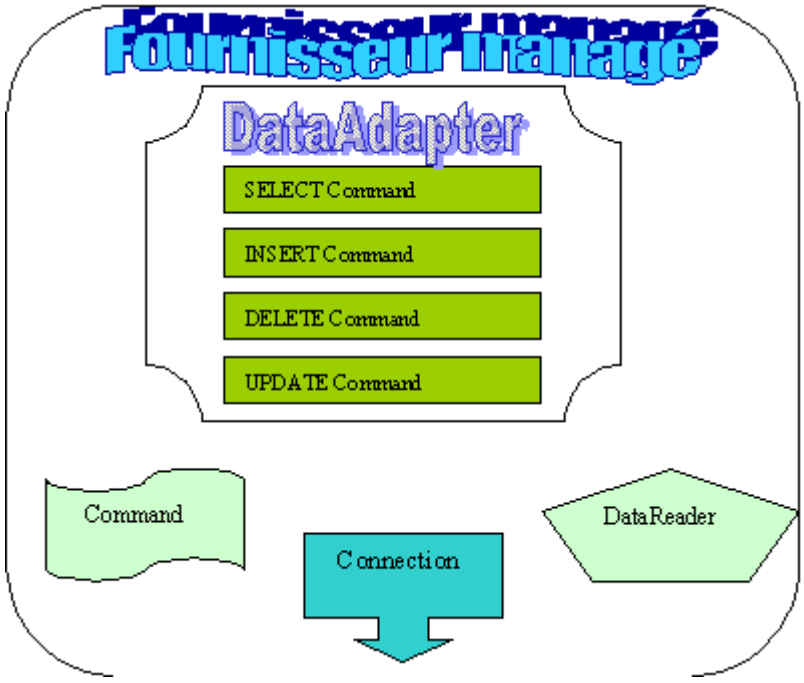


L'apport d'un tel modèle par rapport au Recordset tant en souplesse qu'en fonctionnalité engendre toute une différence de philosophie dans l'approche du développeur. Dans le modèle ADO, on programme avec les limitations induites par l'objet Recordset, avec ADO.NET, on peut se concentrer plus sur les 'points techniques' de l'application (économie de bande passante, études des points de concurrence, etc…).

Il est pourtant faux de croire que la programmation ADO.NET est plus simple car elle demande une meilleure connaissance du fonctionnement d'un SGBD et de bonnes bases de SQL. Ceci s'entend au terme de la compréhension, car l'environnement de développement va travailler beaucoup plus que nous quant à l'écriture du code.

Par de nombreux aspects cependant, la programmation d'ADO et d'ADO.NET est similaire. Vous allez voir tout au long de ce cours que nous allons manipuler des concepts relativement simples ; les seules véritables difficultés étant dues à la programmation des bases de données.

Fournisseur managé



C'est le deuxième concept amené par ADO.NET. Ceci consiste à regrouper dans un fournisseur suivant les règles du FrameWork l'ensemble des classes pouvant communiquer avec la source de données. Là encore si vous avez bien saisi le fonctionnement d'ADO, vous allez voir que le terrain est connu.

Objet Connection

Sensiblement identique à une connexion ADO. Pour ouvrir la connexion on passe une chaîne de connexion à l'objet Connection puis on ouvre celui-ci.

Par exemple pour ouvrir une connexion SQL-Server

```
String strConn = "packet size=4096;user id=sa;data source=NOM-
SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=northwind;password=monpasse";
   System.Data.SqlClient.SqlConnection MaConn = New
System.Data.SqlClient.SqlConnection(strConn);
   MaConn.Open();
```

Pour une base Access il faut utiliser le fournisseur managé pour OLE DB

```
string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data
source=C:\\tutoriel\\dotnet\\biblio.mdb";
OleDbConnection conn = new OleDbConnection(strConn);
try
{    Conn.Open();}
catch(Exception ex)
{    MessageBox.Show(ex.Message);}
finally
{    Conn.Close();}
```

Si vous regardez dans la documentation les propriétés / méthodes de la classe vous allez voir qu'il s'agit de la même connexion qu'ADO, à ceci près qu'il existe un objet Connection différent par fournisseurs managés.

Objet Command

Le même que pour ADO. Gère les paramètres comme son homonyme et s'utilise de la même façon. Il possède quatre méthodes Execute selon qu'il devra créer un objet DataReader, un objet XMLReader, récupérer une valeur ou ne rien récupérer. Il est à noter que l'apparition de la méthode ExecuteScalar est vraiment un plus notable. En effet dans ADO, pour récupérer la valeur renvoyée par la requête :

```
SELECT Count(Description) AS NbDesc FROM Catégories;
```

Il fallait créer un Recordset pour aller lire la valeur, avec tout ce que cela implique. Maintenant un appel à la méthode ExecuteScalar suffit.

Notons que la gestion des paramètres est elle aussi identique. Par exemple, le code ci dessous est du C# utilisant ADO.NET

```
String strConn = "packet size=4096;user id=sa;data source=NOM-
SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=northwind;password=monpasse";
SqlConnection MaConn = New SqlConnection(strConn);
MaConn.Open();
SqlCommand cmd = New SqlCommand("byroyalty", MaConn);
cmd.CommandType = CommandType.StoredProcedure;
SqlParameter par = New SqlParameter("@percentage", SqlDbType.Int);
par.Direction = ParameterDirection.Input;
par.Value = 15;
cmd.Parameters.Add(par);
SqlDataReader dr = cmd.ExecuteReader()
```

Avouons qu'il y a presque de quoi se tromper.

Objet DataReader

Bien que semblant nouveau, tous les développeurs ADO l'ont utilisé de nombreuses fois. En effet il ne s'agit de rien d'autre que du FireHose Cursor, le curseur par défaut d'ADO.

C'est le seul objet 'connecté' fourni par ADO.NET. Comme il s'agit d'un curseur parfaitement connu je ne vais pas m'étendre dessus, toutefois je vous rappelle qu'il a pour propriétés :

- Lecture seule
- En avant seulement
- Un enregistrement à la fois
- Exclusif sur sa connexion

```
{
string  strConn = "packet size=4096;user id=sa;data source=NOM-
SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=northwind;password=monpasse";
SqlConnection MaConn = New SqlConnection(strConn);
MaConn.Open();
SqlCommand cd = New SqlCommand("select * from authors", MaConn);
SqlDataReader rd = cd.ExecuteReader();
while (rd.Read())
     {Console.WriteLine(rd("au_lname"));}
rd.Close();
MaConn.Close();
}
```

Comme cela il est plus facile à reconnaître. Vous noterez au passage que l'on utilise plus la méthode MoveNext pour naviguer à l'intèrieur mais la méthode Read (celle-ci revient à une equivalence de MoveNext et EOF).

L'objet DataAdapter

Voilà par contre un petit nouveau, du moins en apparence. En fait c'est une autre vision de notre vieil ami le moteur de curseur client. Je m'explique.

Dans ADO, lorsqu'on travaille sur un Recordset, on utilise une commande, explicite ou non pour obtenir les enregistrements. Pour transmettre les modifications apportées au Recordset vers la source, le moteur de curseur génère les commandes SQL équivalentes (ou du moins essaye-t-il). Lorsqu'on travaille en mode par lots, il parcours toutes les lignes modifiées et génère l'ensemble des requêtes nécessaires. L'objet DataAdapter regroupe les fonctionnalités de la commande de création et des modifications vers la source. Par contre la stratégie de modification des données a été étoffée.

Vous avez maintenant le choix entre laisser le DataAdapter générer seul les commandes par le biais de son CommandBuilder, ou vous pouvez écrire des commandes génériques pour les requêtes actions de modification. Comme nous allons y revenir longuement dans la troisième partie je ne vais pas m'étendre ici sur le sujet. Néanmoins il faut bien voir qu'il y a là une petite révolution.

Avec ADO, le moteur de curseur client était rarement capable de générer des modifications sur plusieurs tables. En écrivant le code générique de la commande vous pouvez aisément contourner cette limitation, pour peu que vos notions de SQL ne soit pas trop limitées.

Le couplage DataAdapter / Dataset donne un environnement de développement cohérent et performant. Néanmoins c'est principalement dans le paramétrage de cet ensemble que l'on peut créer une application véritablement adaptée. Sauf dans le cas d'application basique, il y a encore nécessité de parfaitement définir les besoins afin d'obtenir les meilleures performances.

Il ne nous reste plus qu'à voir l'intégration du XML dans ADO.NET, et nous avons fini de disséquer la créature. Comme je vous l'avais dit, rien de bien spectaculaire. Par contre nous disposons maintenant d'un ensemble de classes robustes pour le travail sur des données déconnectées.

Intégration du XML

Le XML (Extensible Markup Language) est un langage de méta-balise qui constitue un format pour décrire des données structurées. XML est en train de devenir le langage universel pour les données sur le Web. XML permet aux développeurs de générer des données structurées à partir d'un grand nombre d'applications directement sur le poste de travail comme sur le WEB.

Document XML

Un DataSet est donc utilisé comme une base de données 'en mémoire'. Cette base de données, donc notre DataSet, peut être rendue persistante sous forme fichier XML en utilisant les dérivées de la classe XMLDocument. Je ne vais pas dans le cadre de ce cours aller très loin dans la manipulation des documents XML, néanmoins il est nécessaire d'avoir un minimum de connaissances pour comprendre les exemples que nous serons amenés à manipuler.

La partie qui va nous intéresser plus particulièrement dans ADO.NET est la classe XMLDataDocument.

NB : On trouve facilement sur le Web les spécifications complètes du XML.

XMLDataDocument

XMLDataDocument dérive de XMLDocument. C'est une liaison entre un Document XML et un Dataset. Dans l'absolu, le document XML et le Dataset vont représenter les mêmes données, même s'il va y avoir des différences dans la façon de les présenter, c'est à dire des différences de vue entre des données relationnelles et des données hiérarchiques. Si vous avez utilisé le fournisseur DataShape dans ADO, vous avez rencontré le même problème. Dans une vision hiérarchique on raisonne vis-à-vis du parent. C'est similaire au parcours d'un arbre. Pour retrouver les données d'un enfant il est nécessaire de redescendre la hiérarchie des parents. Le modèle relationnel est quand à lui beaucoup plus à plat. Les enfants sont accessibles et contiennent les informations de leurs parents qui dans une certaine mesure sont dupliquées. Un document XML tend à donner une vision hiérarchique des données, tandis qu'un Dataset présente une vue relationnelle. Un document XMLData tend à supprimer cette différence de vue entre les deux, on peut dire qu'il s'agit d'un document XML sachant exposer ses données dans un Dataset.

Un document XMLData peut être converti vers ou à partir d'un DataSet directement. Il est possible de synchroniser les deux pour que toutes modifications de l'un se répercutent vers l'autre.

Conversion XMLDocument ⇔ XMLDataDocument

La conversion vers un XMLDataDocument ne pose pas de problème majeur. En fait, il y a conversion directe sachant que les données non relationnelles risquent d'être supprimées. La conversion dans l'autre sens est un plus complexe et fait appel à des notions de feuille de style (XSL). Je ne l'utiliserais pas dans le présent cours.

Nous verrons dans la troisième partie comment manipuler ces documents XML avec des DataSet.

XML Schema (XSD)

Pour pouvoir passer aisément des données d'un Dataset à celle d'un document XML, vous devez pouvoir transmettre des informations de schéma. Pour cela on utilise le langage XSD (XML Schema Definition). Du fait de la complexité possible d'un schéma de données dans un Dataset, il peut être très complexe d'écrire le schéma. A voir ainsi, cela semble assez abscons mais du fait de la forte intégration des deux modèles c'est très simple à utiliser. En effet vous n'aurez jamais besoin de travailler en profondeur sur ces modèles objets. ADO.NET implémente toutes les méthodes nécessaires pour travailler simultanément sur des documents / schémas XML et sur vos DataSet, ainsi que toutes les méthodes nécessaires pour passer de l'un à l'autre.

Un même Dataset peut donner plusieurs schéma différents selon son paramétrage, il peut être alors interessant de communiquer le schéma avant ou sans les données selon les cas.

Utiliser ADO avec les langages DotNet

Pour tordre le cou à une des légendes souvent citées, nous allons voir dans le petit exemple suivant comment utiliser ADO pour générer un Recordset et transformer celui-ci en Dataset. Ceci nous permettra de discuter dans ce chapitre sur l'intérêt de continuer d'utiliser ADO avec C# et de quand et comment le gérer.

Pour faire fonctionner cet exemple, vous devez intégrer une référence à ADODB dans l'onglet COM.

```
{
String MaConn="PROVIDER=SQLOLEDB;user id=sa;data source=NOM-
SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=northwind;password=monpasse";
ADODB.Recordset MonRst=new ADODB.Recordset();
MonRst.CursorLocation=ADODB.CursorLocationEnum.adUseClient;
MonRst.Open("SELECT * FROM produits",
MaConn,ADODB.CursorTypeEnum.adOpenStatic,ADODB.LockTypeEnum.adLockBatchOp
timistic,0);
OleDbDataAdapter MonDtaAdapter = new OleDbDataAdapter();
DataSet MonDataset = new DataSet();
MonDtaAdapter.Fill(MonDataset, MonRst, "produits");
dataGrid1.DataSource=MonDataset.Tables[0].DefaultView;
}
```

Comme vous le voyez il s'agit d'une programmation triviale. Dans ce cadre elle est en plus inutile car utiliser ADO pour générer un Recordset déconnecté lorsqu'on peut utiliser ADO.NET c'est du vice.

Pourquoi utiliser encore ADO ?

Comme nous l'avons vu, ADO.NET est une extension d'ADO. Cette extension à pour but de gérer une partie par trop limitée du modèle ADO. Néanmoins, il existe un domaine ou ADO.NET ne peut pas être utilisé, le travail côté serveur. Comme nous l'avons vu, ADO.NET ne met à notre disposition qu'un seul curseur serveur, le DataReader. Si ces avantages en terme de performances sont évidents, ses limitations le sont tout aussi. Impossible de verrouiller un enregistrement puisque le curseur est en lecture seule et pas de sensibilité aux données sur les enregistrements déjà lus.

Les limitations de la concurrence optimiste

Pour avoir une vue de plusieurs enregistrements dans ADO.NET, pas d'autres choix que le Dataset. Rappelons-nous bien que celui-ci est un curseur statique, optimiste par lot, côté client. Il est d'autant plus statique qu'il est déconnecté. Il serait techniquement possible d'envisager un rafraîchissement périodique d'un Dataset en le remplissant régulièrement mais le coût en temps/ressources rend cette hypothèse inabordable.

Lorsqu'un Dataset est modifié et qu'on lui demande de répercuter ses modifications vers la source de données, celui-ci gère la concurrence par un verrouillage optimiste.

Par définition le verrouillage optimiste n'est pas un verrou mais une comparaison de valeurs. Il existe grosso modo deux gestions possibles.

Récupération d'une valeur de version : Ceci consiste à utiliser une donnée qui est intrinsèque à la ligne mais qui n'est pas une donnée de l'enregistrement. Par exemple une valeur TimeStamp de la dernière modification. Lors de l'appel de la mise à jour, il y a comparaison de la valeur de l'enregistrement et de celle stockée dans le Dataset. Si elles sont différentes, c'est à dire si un autre utilisateur a modifié l'enregistrement, il y a conflit de la concurrence optimiste, une exception est générée et la mise à jour est refusée. Ce mode est géré par SQL-Server par exemple.

Comparaison des enregistrements : C'est le principe utilisé par le moteur de curseur client. Elle consiste à comparer les valeurs des champs trouvées dans le Dataset à celles présentes dans la source. Elle ne porte pas forcément sur l'ensemble des valeurs. Le principe réel est simple on exécute une requête action contenant dans la clause WHERE les valeurs de champs présentes dans le Dataset, et on récupère le nombre d'enregistrements modifiés. Si celui-ci est nul, c'est que l'enregistrement a été modifié et qu'il y a concurrence optimiste. Cette technique peut être périlleuse si on n'utilise pas tous les champs dans la requête, mais c'est une question de choix.

Il existe aussi une méthode de barbare que les anglophones nomment 'last in win' qui consiste à ne pas gérer la concurrence et à toujours écraser l'enregistrement par les dernières modifications arrivant. Je vous laisse concevoir quels peuvent être les effets de ce genre de stratégie.

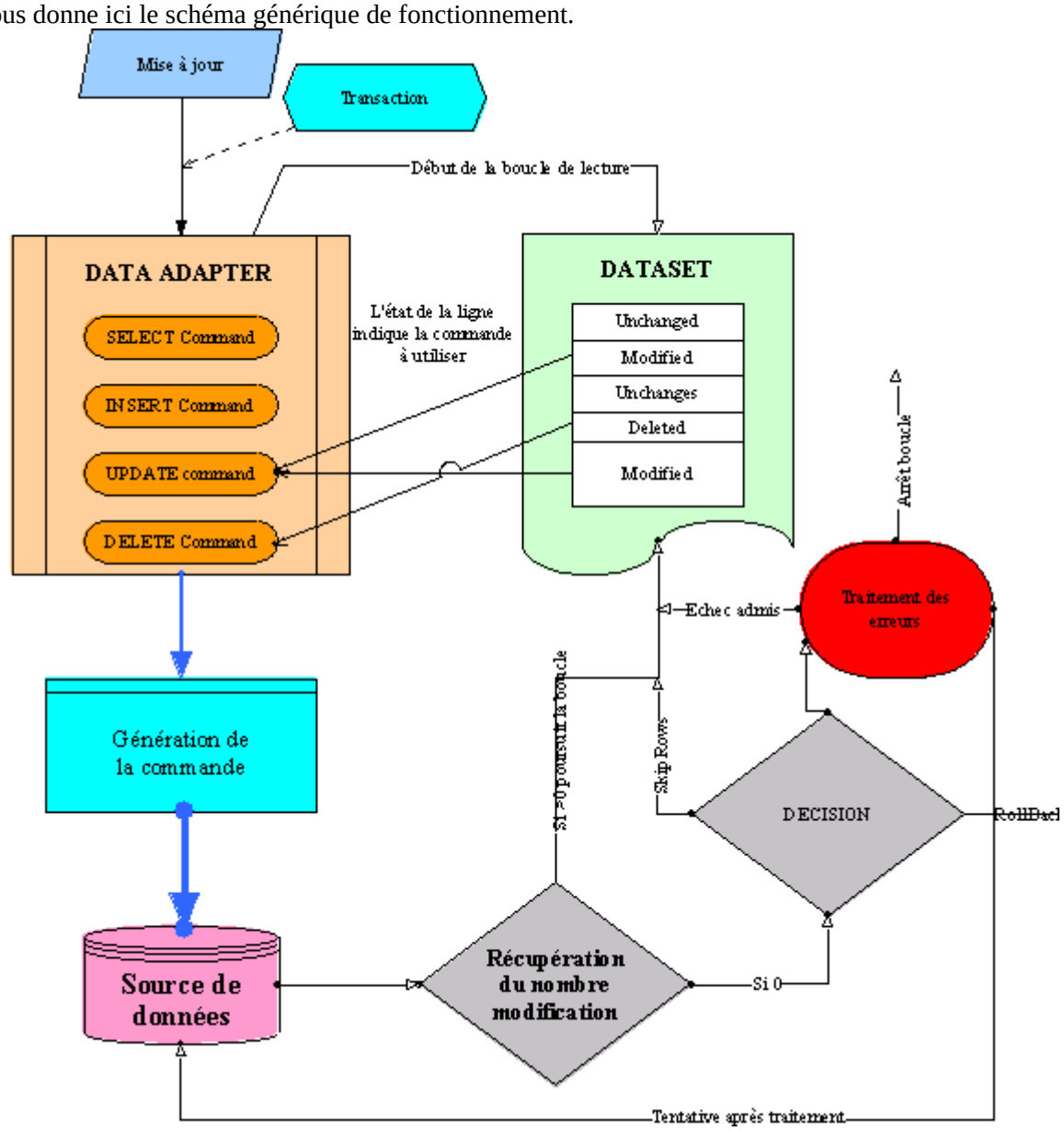
Dès lors qu'il y a concurrence optimiste, il y a globalement deux réponses possibles. On travaille dans une logique de transaction, comme une modification ne peut avoir lieu, toutes les modifications doivent être rejetées, et on fait alors un RollBack de la transaction. On travaille en mode 'sans unité' et on valide les modifications qui sont acceptées. Libre ensuite de reprendre ou non après un traitement sur les enregistrements refusés pour tenter de leur adapter une nouvelle modification.

La stratégie que l'on applique pour la concurrence doit être définie en toute connaissance de cause. En choisir une au hasard revient à ne pas gérer la concurrence.

Il est souvent préférable de travailler sur des opérations de petites tailles et cohérentes. En effet l'approche transactionnelle est beaucoup plus fiable et facile à gérer que l'acceptation partielle avec traitement des erreurs.

Enfin ayez bien à l'esprit que la simple gestion de la concurrence ne suffit pas à garantir l'intégrité des données. Dans notre cas par exemple vous voyez bien qu'il n'y a pas de sécurité au niveau de la concurrence stricto sensu pour l'ajout d'une nouvelle ligne.

Je vous donne ici le schéma générique de fonctionnement.



Ce que nous voyons bien avec ce type de verrouillage est la limitation du modèle. A aucun moment je n'ai un véritable contrôle sur ce que sera ma mise à jour. Soit j'utilise une transaction, et l'échec d'une modification entraînera l'échec de toute la transaction, soit je gère moi-même le comportement en cas d'erreur sachant qu'il y aura peut-être alors des modifications rejetées.

La seule façon de pouvoir exercer un contrôle strict sur mes données consiste donc à utiliser un curseur serveur.

Les contraintes d'ADO

Le choix d'ADO ne peut être fait que si le modèle COM est supporté par toutes les parties. De plus, il n'est réaliste que dans une application de type client / Serveur. Dans les langages DotNet les performances vont en prendre un coup du fait de l'utilisation de l'interop. Enfin et surtout n'oubliez pas que l'utilisation côté serveur est consommatrice de ressources serveur et qu'il faut que celui-ci puisse tenir la charge. Ce type d'applications n'est pas fréquente, on le retrouve en général dans deux grandes familles :

Les applications à concurrence élevée : C'est le cas des applications ou l'accès aux données est extrêmement fréquent et ou la conception de la source fait que le risque de point de concurrence a une forte probabilité. Il faut avant tout évaluer si le coût, dans le sens large du terme, rend rentable d'utiliser un curseur serveur plutôt que de faire des RollBack sur de nombreuses transactions.

Les applications à affichage dynamique : Ce sont celles qui demandent que les contrôles consommateurs reflètent toujours l'état réel des données. Ce genre d'application demande une écriture particulièrement soignée afin que le serveur puisse supporter la charge.

Comment utiliser ADO ?

L'opération est assez simple. Tout d'abord on sélectionne le menu "projet-Ajouter une référence" pour ajouter une référence à ADODB (Microsoft® ActiveX Data Object 2.7 +). Celle-ci se trouve dans l'onglet COM.

Dans le menu projet, allez dans les propriétés du projet et importez l'espace de nom System.Data.

```
private void WinForm_Load(object sender, System.EventArgs e)
{
String MaConn="PROVIDER=SQLOLEDB;user id=sa;data source=NOM-
SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=northwind;password=monpasse";
ADODB.Recordset MonRst=new ADODB.Recordset();
MonRst.CursorLocation=ADODB.CursorLocationEnum.adUseClient;
MonRst.Open("SELECT * FROM produits",
MaConn,ADODB.CursorTypeEnum.adOpenStatic,ADODB.LockTypeEnum.adLockBatchOp
timistic,0);
OleDbDataAdapter MonDtaAdapter = new OleDbDataAdapter();
DataSet MonDataset = new DataSet();
MonDtaAdapter.Fill(MonDataset, MonRst, "produits");
dataGrid1.DataSource=MonDataset.Tables[0].DefaultView;
}
```

Dans ce cas j'utilise un Datagrid. Ceci pose un problème évident. En effet, l'utilisation de ce contrôle me force à convertir mon Recordset en Dataset, ceci me faisant perdre l'actualisation des données. Pour ne pas rencontrer ce problème, je dois construire un formulaire différent et lier mes données dynamiquement.

Il va être important de régler une taille de cache assez petite, puisque lors des rafraîchissements du curseur, les enregistrements présents dans le cache ne sont jamais rafraîchis.

Dans cet exemple nous allons faire une visionneuse d'enregistrements.

```
private void WinForm_Load(object sender, System.EventArgs e)
      {
      string MaConn = "Provider=Microsoft.Jet.OLEDB.4.0;User
ID=Admin;Data Source=C:\\tutoriel\\biblio.mdb;";
      MonRst.Open("SELECT * FROM Authors", MaConn,
ADODB.CursorTypeEnum.adOpenStatic,
ADODB.LockTypeEnum.adLockBatchOptimistic, 0);
      txtId.Text = Convert.ToString(MonRst.Fields[0].Value);
      txtAut.Text = Convert.ToString(MonRst.Fields[1].Value);
      txtYear.Text = Convert.ToString(MonRst.Fields[2].Value);
      }

void MonRst_MoveComplete(ADODB.EventReasonEnum a, ADODB.Error b, ref
ADODB.EventStatusEnum c, ADODB.Recordset d)
      {
      txtId.Text = Convert.ToString(MonRst.Fields[0].Value);
      txtAut.Text = Convert.ToString(MonRst.Fields[1].Value);
      txtYear.Text = Convert.ToString(MonRst.Fields[2].Value);
      }

private void cmdFirst_Click(object sender, System.EventArgs e)
      {
            MonRst.MoveFirst;
      }

private void cmdNext_Click(object sender, System.EventArgs e)
      {
            MonRst.MoveNext;
      }

private void cmdPrev_Click(object sender, System.EventArgs e)
      {
            MonRst.MovePrevious;
      }

private void cmdLast_Click(object sender, System.EventArgs e)
      {
            MonRst.MoveLast;
      }
```

Cette simple visionneuse me permet aussi de modifier les valeurs de mes enregistrements.

Comme nous le voyons cela se programme très facilement, et nous avons construit une application connectée, visualisant les modifications des autres utilisateurs et gérant des verrous physiques.

Considérations sur les performances

Vous allez trouver de nombreux comparatifs sur les performances entre ADO et ADO.NET. Ces 'tests' montrent très souvent des différences colossales. Il est certes facile de produire ces différences, mais très souvent on omet d'en donner les raisons, ce qui fait que l'intérêt de ces tests n'est pas bien grand. Cela est d'autant plus vrai que l'on peut produire les mêmes différences entre deux applications utilisant ADO.NET.

Informations de schéma

Lorsqu'on travaille avec ADO, on délègue un certain nombre de taches aux divers fournisseurs de services. C'est la puissance de ceux-ci qui permettent une programmation très simple mais il y a un coût en terme de performance. Lorsqu'on crée un Recordset côté client, le moteur de curseur gère la récupération des informations de schéma. Globalement cela permet une programmation plus générique et limite les connaissances nécessaires en programmation des SGBD. Malheureusement cette récupération se produit à l'exécution et se produit pour chaque objet Recordset. Plus le schéma est complexe, plus les opérations sont lourdes et il n'est plus possible d'optimiser passé un certain point.

Création ou exécution

Si avec ADO il n'y a pas de gain de temps notable lorsqu'on connaît la source à la création, avec ADO.NET cela peut changer énormément la façon d'aborder la programmation. On peut décider, en échange d'un travail à la création de lier son application à la source afin d'économiser du temps d'exécution. Bien sur cela n'a pas lieu d'être lors de la fabrication d'une application générique, mais dans la plupart des applications cela peut augmenter fortement les performances.

Récupération du schéma à l'exécution

La récupération des informations de schéma est coûteuse lorsqu'elle est dynamique. Dans ADO.NET, il y a plusieurs façon de gérer les méta-données de la source. La méthode FillSchema du DataAdapter est dynamique mais nuit aux performances, utiliser des schémas XSD est un meilleur compromis. Travailler sur un schéma défini à la conception permet un gain de performance contre une perte de souplesse.

Dataset fortement typé

Il s'agit de créer une classe dérivée du Dataset autour d'un schéma défini. Vous pouvez soit partir de l'utilitaire xsd.exe et d'un fichier de schéma XSD, soit utiliser l'environnement de développement, soit générer vous-même la classe. Ce type de Dataset perd une grande part de sa souplesse puisqu'il est construit autour d'une source définie. Toute modification de structure à l'exécution induit un travail important de régénération. Par contre, il y a un gain important dans la qualité du code développé en utilisant de tels Dataset. Le gain en terme de performances est quant à lui du à l'utilisation de DataColumn plutôt que de nom de champs, ce qui fait qu'il n'est pas nécessaire d'utiliser un Dataset fortement typé pour obtenir de bonnes performances. Nous verrons un exemple de ces Dataset dans l'exemple "Une base de donnée locale".

Désactivation des contraintes

Quel que soit le cas, votre Dataset possède généralement des contraintes, souvent récupérées de la source de données, dans certains cas créées par vous. Lors du remplissage du Dataset par les données de la source, vous devez veiller à désactiver les contraintes importées de la source. En effet, chaque contrainte induit une vérification de l'enregistrement importé. Cette vérification coûteuse en performances est inutile puisque les données de la source respectent déjà ces contraintes. Il existe plusieurs techniques que nous étudierons pour éviter cette redondance du contrôle.

Vous ne pouvez pas procéder ainsi pour les contraintes que vous avez ajoutées.

Gestion des noms de champs

Il s'agit là d'un problème assez similaire à l'utilisation des objets Fields dans ADO. Il existe trois méthodes pour désigner un champ dans un enregistrement.

Par le nom du champ :

```
txtAuthor.Text = MonDs.Tables[0].Rows[0]["Author"];
```

Cela donne un code lisible et assez polyvalent mais de très mauvaises performances.

Par la position ordinale

```
txtAuthor.Text = MonDs.Tables[0].Rows[0][1];
```

Les performances sont bien meilleures, mais la position ordinale dépend du texte de la requête et la lisibilité du code est médiocre.

Par l'objet DataColumn

```
txtAuthor.Text =
MonDs.Tables[0].Rows[0](MonDs.Tables[0].Columns["Author"]);
```

La notation est plus lourde mais les performances sont excellentes.

L'utilisation de la première syntaxe devrait être bannie de vos applications car elle n'est jamais rentable. Je vous montrerai plus loin comment utiliser des codes mariant performances et lisibilité.

Programmation des SGBD administrés

Nous allons ici envisager un cas qui devrait être le cas général, bien que ce soit loin d'être encore le cas. Il y a deux grandes familles de SGBD dont on ne parle jamais, celle que vous administrez et celle qui est administrée par quelqu'un d'autre.

Administrer une base de données est un métier et de nombreux développeurs ignorent les rudiments de cette profession. Cela tend à les faire déchanter lorsqu'ils doivent programmer une base de données qu'ils n'ont pas créée.

Je vais donc parler ici des règles couramment admises dans le développement d'applications autour de SGBD administrés, en espérant que cela vous donnera les idées claires quand il vous faudra aller négocier avec l'administrateur.

Propriétés des objets et login

La plupart des SGBD définissent un propriétaire pour les tables, requêtes, vues, etc.….

Ce propriétaire possède les droits de modification et de suppression pour ses objets. Habituellement, l'administrateur est le propriétaire des principaux objets. Il restreint aussi la possibilité de créer des objets et / ou d'y accéder.

Comme les restrictions peuvent être plus ou moins importantes, c'est par l'utilisation du Login qu'il définit les droits de chaque utilisateur. Il vous faut donc étudier les droits **véritablement** nécessaires à votre application avant de négocier la création d'un login adapté à votre application. Il est de toute façon peu probable que tout les droits vous seront attribués, il convient donc d'analyser correctement le besoin.

Requêtes et procédures stockées

Dans la plupart des cas, seules les requêtes SELECT sont autorisées librement. L'ensemble des requêtes actions est généralement interdit par un appel SQL direct, et l'administrateur vous demandera d'agir sur la base soit par l'appel de procédures et/ou de requêtes stockées existantes, soit en lui fournissant la liste des procédures / requêtes que vous voulez qu'il ajoute dans la source.

Dans certains cas, même la liste des vues est restreinte.

Si cette démarche est tout à fait légitime de la part de l'administrateur cela peut vous causer des problèmes lors du développement de vos applications. La grande majorité des administrateurs sont des gens intelligents, commencez donc par aller discuter avec eux afin de connaître les règles qu'ils souhaitent voir appliquées ainsi que les contraintes qu'il vous semble impossible de respecter.

Lorsqu'il ne s'agit que d'utiliser des requêtes stockées, les contraintes ne sont pas bien grandes, par contre il peut en aller tout autrement pour les procédures stockées.

Considérations sur les procédures stockées

Une procédure stockée est une suite de commandes écrites dans un langage supporté par le SGBD. Cette série d'instructions peut être vue comme plusieurs ordres SQL consécutifs du côté du programmeur. Sachez que tous les SGBD ne supportent pas les mêmes fonctionnalités, dès lors votre codage peut être différent selon la source de données.

Il n'est pas réaliste d'utiliser une procédure stockée sans l'avoir lue et correctement comprise. Globalement voilà les problèmes que l'on peut rencontrer.

Valeurs de compteur non retournées

Certaines tables utilisent un champ dit 'auto-incrémenté' pour générer la valeur de la clé primaire. Cette valeur est fournie par le SGBD. Vous avez besoin de cette valeur pour créer des enregistrements liés. Habituellement un paramètre de sortie renvoie cette valeur, mais certains administrateurs préfèrent renvoyer la ligne insérée. Il est aussi possible que la procédure soit mal rédigée et que la valeur ne soit pas renvoyée.

Nombre de lignes modifiées faux

On utilise ce nombre pour savoir la portée de la procédure. Il arrive qu'une procédure mal écrite ne renvoie pas le nombres de lignes affectées. Il arrive parfois aussi que le compte intègre les lignes créées pour l'administration.

Mise à jour différée

Dans quelques rares cas, les modifications attendent une validation de l'administrateur avant d'être véritablement acceptée par la source. Ceci peut être transparent si le système est bien géré, mais peut aussi engendrer des problèmes quasiment insurmontables. Il convient alors de voir avec l'administrateur les point de blocage que vous rencontrez.

Lot de requêtes

Certains SGBD renvoient des lots de requêtes. Ceux-ci vont vous forcer à une gestion particulière dans votre code afin de ne pas polluer votre Dataset et votre code décisionnel.

Gestion des erreurs du SGBD

Votre application ne pourra jamais faire de l'administration. Il faut clairement comprendre que les erreurs que votre code peut récupérer sont des erreurs vénielles pour le SGBD. Si celui rencontre de graves erreurs il fermera votre connexion. Ce cas doit être envisagé mais il est sans solution dans le cadre d'une application cliente.

Les erreurs engendrées par votre code sont très faciles à corriger sur un SGBD administré. Elles peuvent l'être beaucoup moins si vous administrez vous-même la base de données et si vous ne fixez aucune règle. Voilà pourquoi je vous ai dit au début que vous devriez procéder de façon identique avec vos propres bases en étant un administrateur consciencieux pour vos propres sources de données.

N'oubliez jamais que l'intégrité des données est la seule règle avec laquelle vous ne pouvez pas transiger.

Les classes ADO.NET

Comme à chaque fois, nous allons visiter en détail les classes fournis par ADO.NET. Dans ce cours je travaillerais principalement autour de la base exemple 'Biblio.mdb' pour Access 2000 et donc avec les classes de l'espace de nom System.Data.OleDB. Je vais privilégier ce fournisseur pour obtenir un code relativement générique. Du fait des limitations d'Access et de la spécificité du fournisseur SQL-Server, un certains nombre d'exemple utiliseront le fournisseur managé SQL-Server et la base exemple 'Pubs' (System.Data.SqlClient).

Comme dans mes autres cours, la liste suivante n'est pas exhaustive, elle n'indique que les propriétés / méthodes utilisées dans une programmation 'standard'. Les propriétés évidentes comme le nom des objets ou les propriétés / méthodes du à l'héritage de la programmation des objets ne seront pas abordés.

Les espaces de nom dans DotNet

Un espace de nom dans DotNet est une organisation de classes connexes. Il s'agit en fait d'une modification de la notion de portée. Dans DotNet chaque objet possède un nom qui appartient à un espace de nom, le nom devant être unique dans l'espace. Il est donc possible d'utiliser deux objets ayant le même nom dans un même assembly pour peu que l'espace de nom soit donné avant l'objet. En général, on ne met qu'un espace de nom dans un assembly et on peut alors utiliser directement l'objet sans donner son espace de nom. Regardons l'exemple suivant :

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn, strSQL;
strConn = "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
System.Data.OleDb.OleDbConnection ConnAccess = new
System.Data.OleDb.OleDbConnection(strConn);
ConnAccess.Open();
strSQL = "SELECT Author FROM Authors WHERE Au_Id=1";
System.Data.OleDb.OleDbCommand CommandAccess = new
System.Data.OleDb.OleDbCommand(strSQL, ConnAccess);
textBox1.Text = CommandAccess.ExecuteScalar().ToString();
}

private void button2_Click(object sender, System.EventArgs e)
{
String strConn, strSQL;
strConn = "User ID=sa;Initial Catalog=pubs;Data Source=NOM-
SZ71L7KTH17\\TUTO;password=monpasse";
System.Data.SqlClient.SqlConnection ConnSQL = new
System.Data.SqlClient.SqlConnection(strConn);
ConnSQL.Open();
strSQL = "SELECT au_lname FROM Authors WHERE Au_Id='172-32-1176'";
System.Data.SqlClient.SqlCommand CommandSQL = new
System.Data.SqlClient.SqlCommand(strSQL, ConnSQL);
textBox2.Text = CommandSQL.ExecuteScalar().ToString();

}
```

Nous voyons donc que des objets similaires possèdent une syntaxe différente ce qui nous permet de travailler avec l'espace de nom System.Data. A part cette syntaxe il n'y a que peu de différences entre des objets définis par les fournisseurs managés, je veux dire que par exemple les objets OLEDBConnection et SqlConnection se manipulent de la même manière.

La classe Dataset

Comme la programmation ADO.NET tourne principalement autour de l'objet Dataset, nous allons commencer par lui. Le Dataset peut donc être vu comme un magasin de données, l'enjeu étant de ne pas le transformer en dépotoir. Un Dataset est un container pour des objets DataTable et DataRelation. Pour schématiser, on dira qu'un Dataset sans données est lié à un fichier de schéma XSD et qu'un Dataset rempli est lié à un fichier XMLDataDocument. Cette liaison passe à travers un certain nombre de méthode qui permettent de pouvoir synchroniser l'un et l'autre. En soi, un Dataset n'a pas d'autre fonction que de véhiculer un schéma.

ExtendedProperties & PropertyCollection

Les objets Dataset, DataColumn, DataTable, DataRelation et Constraint possèdent une collection PropertyCollection qui permet à l'utilisateur de rajouter des propriétés personnalisées à ces objets.

Ces propriétés s'utilisent comme n'importe quelle propriété. Elles doivent être de type String pour être transmises au fichier XML.

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn, strSQL;
strConn = "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
strSQL = "SELECT * FROM Authors";
OleDbDataAdapter MonDA As = OleDbDataAdapter(strSQL, strConn);
DataSet MonDataset = New DataSet("Auteurs");
MonDA.Fill(MonDataset);
MonDataset.ExtendedProperties.Add("TimeStamp",DateTime.Now);
Me.DataGrid1.DataSource = MonDataset.Tables[0];
}

private void dataGrid1_DataSourceChanged(object sender, System.EventArgs
e)
{
DataSet UnDataset = CType(Me.DataGrid1.DataSource.Dataset, DataSet);
MessageBox.Show(UnDataset.ExtendedProperties("TimeStamp").ToString());
}
```

Propriétés

CaseSensitive (Booléen)

Définit si les objets DataTable qui composent le Dataset sont sensibles à la casse lors des opérations de recherche, de tri et de filtrage. Comme cette propriété existe aussi pour l'objet DataTable, la définition au niveau du Dataset revient à la définition de la propriété pour chacun des DataTable.

DefaultViewManager (DataViewManager)

Renvoie le DataViewManager utilisé par défaut pour gérer les vues du Dataset. Nous verrons le fonctionnement des vues un peu plus loin.

EnforceConstraints (Booléen)

Définit si les contraintes doivent être appliquées. Typiquement, on désactive les contraintes lors de la récupération des données d'une source ou celles-ci ont déjà été validées. Certain développeurs préfèrent rapatrier les données puis ajouter les contraintes mais cela n'a pas véritablement de sens à mes yeux. En effet, si l'on procède à une mise à jour d'un Dataset existant, il paraîtrait incohérent de supprimer toutes les contraintes pour les rétablir après la récupération.

Lors du rétablissement des contraintes, il y aura levée d'une exception si des données ne les respectent pas. On peut se demander alors s'il est intéressant de les désactiver puisque le contrôle a lieu de toute façon. Cependant, ce contrôle est beaucoup plus rapide sur les lignes du Dataset dans le cache client que lors d'un contrôle pendant la récupération ligne par ligne.

Les lignes violant les contraintes peuvent être récupérées à l'aide de la méthode GetErrors.

HasErrors (Booléen)

Renvoie vrai si une DataTable du Dataset possède une ligne au moins en erreur. Cette propriété existe aussi au niveau de la table.

```
void VerifErreur(DataSet MyDataset)
{
if (MyDataset.HasErrors)
      {
      DataRow[] LigneErreur;
      foreach (DataTable MaTable in MyDataset.Tables)
      if (MaTable.HasErrors)
            LigneErreur  = MaTable.GetErrors();
      }
```

}Locale (CultureInfo)

Définit les informations de langue utilisées par le tri.

Relations (DataRelationCollection)

Obtient la collection des relations qui relient des tables et permettent de naviguer des tables parentes aux tables enfants.

Tables (DataTableCollection)

Obtient la collection des tables contenues dans le DataSet.

Méthodes

AcceptChanges

Valide toutes les modifications en cours. La même méthode existe pour les objets DataTable et DataRow. L'appel au niveau du Dataset bascule toutes les lignes de toutes les tables du Dataset vers l'état 'Unchanged'.

Clear

Supprime toutes les lignes de toutes les tables du Dataset. Lève une exception si le Dataset est lié à un XMLDataDocument. Dans ce cas vous devez supprimer les lignes une à une.

Clone

Renvoie une copie du schéma dans un nouveau Dataset. Les données ne sont pas copiées.

Copy

Renvoie une copie du Dataset dans un nouveau Dataset. Les données sont copiées.

GetChanges

Envoie une copie du Dataset contenant toutes les lignes en cours de modification. Accepte une surcharge permettant de limiter le duplicata en fonction de la valeur DataRowState. Les relations peuvent entraîner la présence de lignes qui ne sont pas en cours de modification dans le nouveau Dataset. Ceci veut dire que lorsqu'une ligne d'une table parente est modifiée, les lignes en relation de la table enfant seront incluses dans le Dataset.

HasChanges (Booléen)

Renvoie vrai si une ou plusieurs lignes sont modifiées.

Merge

Permet de fusionner deux Dataset. Il est très important de bien comprendre comment fonctionne cette méthode, aussi allons nous en parler assez souvent tout au long de ce cours. ADO.NET cherche constamment à réussir la fusion quel que soit le scénario. S'il n'est pas simple de faire échouer la fusion, il est par contre assez simple d'obtenir un résultat différent de celui escompté.

La fusion procède toujours par identification des noms locaux. C'est à dire que lorsqu'un objet existe dans un Dataset mais pas dans l'autre, il y a ajout de l'objet à la collection dans le Dataset fusionné. Par exemple:

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
string strSQL = "SELECT * FROM Authors";
OleDbDataAdapter MonDA = new OleDbDataAdapter(strSQL, strConn);
DataSet MonDataset = new DataSet("Auteurs");
MonDA.Fill(MonDataset);
DataSet DatasetCible = new DataSet();
DatasetCible.MergeFailed += new
MergeFailedEventHandler(DatasetCible_MergeFailed);
MonDA.Fill(DatasetCible, "Authors");
DatasetCible.Merge(MonDataset);
foreach (DataTable maTable in DatasetCible.Tables)
     Console.WriteLine(maTable.TableName);
}

void DatasetCible_MergeFailed(Object sender, MergeFailedEventArgs e)
{
MessageBox.Show("La fusion a échoué");
}
```

Bien que stricto sensu j'ajoute deux fois la même table, le Dataset résultant contient deux tables appelées 'Authors' et 'Table' ; nous verrons le pourquoi dans l'étude du mappage des tables par le DataAdapter.

Il y a là un premier danger car par défaut, un DataAdapter donne toujours le même nom aux objets mappés. Imaginons le cas suivant :

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
string strSQL = "SELECT * FROM Authors";
OleDbDataAdapter MonDA = new OleDbDataAdapter(strSQL, strConn);
DataSet MonDataset = new DataSet("Auteurs");
MonDA.Fill(MonDataset);
DataSet DatasetCible = new DataSet();
DatasetCible.MergeFailed += new
MergeFailedEventHandler(DatasetCible_MergeFailed);
MonDA.SelectCommand.CommandText = "SELECT * FROM Publishers";
MonDA.Fill(DatasetCible);
DatasetCible.Merge(MonDataset);
dataGrid1.DataSource = DatasetCible.Tables(0);
}
```

Dans ce cas, je construis deux Dataset contenant chacun une table différente. Lors de la fusion, les deux tables s'appellent 'table' dans leur Dataset respectif. Comme il y a identité de noms, il y a fusion en une seule table. Je crée une table qui contient tous les champs de chaque table et tous les enregistrements. Comme il n'y à pas d'identité, tous les champs appartenant à une table ont une valeur NULL pour les enregistrements de l'autre table. Ceci arrive aussi car il n'y a pas de contraintes sur mes tables. Selon les cas, la situation peut rapidement se complexifier. Si mon code de remplissage est :

```
MonDA.FillSchema(DatasetCible, SchemaType.Source);
MonDA.Fill(DatasetCible);
DatasetCible.Merge(MonDataset);
```

Je vais obtenir une exception de type ArgumentNullException, car la table du dataset DatasetCible possède une clé primaire. Comme il n'y a pas de possibilité d'ajouter une valeur NULL dans un champ de la clé primaire, une exception est levée. Par contre si mon code est:

```
MonDA.Fill(MonDataset);
MonDataset.Tables[0].Constraints.Add("pk1",
MonDataset.Tables[0].Columns[0],true);
DataSet DatasetCible = new DataSet();
DatasetCible.MergeFailed += new
MergeFailedEventHandler(DatasetCible_MergeFailed);
MonDA.SelectCommand.CommandText = "SELECT * FROM Publishers";
MonDA.Fill(DatasetCible);
DatasetCible.Merge(MonDataset);
```

C'est une ConstraintException qui va être levée.

Jusque là, bien qu'ayant toujours déclaré l'événement MergeFailed, il ne c'est jamais déclenché. Ceci vient du fait que la fusion est considérée comme réussie lorsque la fusion des schémas est réalisable.

Je vais maintenant mélanger mes deux codes précédents.

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
string strSQL = "SELECT * FROM Authors";
OleDbDataAdapter MonDA = new OleDbDataAdapter(strSQL, strConn);
DataSet MonDataset = new DataSet("Auteurs");
MonDA.Fill(MonDataset);
MonDataset.Tables[0].Constraints.Add("pk1",
MonDataset.Tables[0].Columns[0],true);
DataSet DatasetCible = new DataSet();
DatasetCible.MergeFailed += new
MergeFailedEventHandler(DatasetCible_MergeFailed);
MonDA.SelectCommand.CommandText = "SELECT * FROM Publishers";
MonDA.FillSchema(DatasetCible, SchemaType.Source);
MonDA.Fill(DatasetCible);
DatasetCible.Merge(MonDataset);
dataGrid1.DataSource = DatasetCible.Tables[0];
}

void DatasetCible_MergeFailed(object sender, MergeFailedEventArgs e)
{
MessageBox.Show("La fusion a échoué");
}
```

L'événement se déclenche car il n'y a pas possibilité de fusionner deux tables portant le même nom mais dont le champ de clé primaire ne porte pas le même nom. Dans les cas précédents, j'obtenais une table contenant tous les champs des deux tables, même si toutes les lignes ne pouvaient pas être fusionnées, dans celui-ci il n'y a pas fusion.

Donc pour fusionner deux tables il faut qu'elles possèdent une clé primaire identique. Dès lors il y a fusion des schémas, puis tentative d'ajout des lignes de la table entrante.

Si une ligne entrante possède la même valeur de clé qu'une ligne existante, les champs communs aux deux tables prennent les valeurs entrantes.

Ce comportement n'est pas toujours le même, il dépend du paramètre MissingSchémaAction. De la même façon, lorsque des modifications sont en cours dans un Dataset, vous pouvez ou non forcer à la conservation de ces modifications. C'est pour cela que je vous conseille d'utiliser la surcharge :

Public void Merge(DataSet ***Dataset,*** Boolean ***PreserveChanges,*** MissingSchemaAction *MissingSchemaAction***);**

Ou MissingSchemaAction peut prendre les valeurs suivantes.

Nom	Description
Add	Ajoute les colonnes nécessaires pour achever le schéma.
AddWithKey	Ajoute les colonnes et les informations de clé primaire nécessaires pour achever le schéma.
Error	Une exception InvalidOperationException est générée si le mappage de colonnes spécifié est manquant.
Ignore	Ignore les colonnes supplémentaires.

Nous reviendrons sur la méthode Merge lors de l'étude des actualisations.

RejectChanges

Rejette l'ensemble des modifications en attente sur toutes les tables du Dataset.

Reset

Réinitialise le Dataset dans son état d'origine.

Propriétés & méthodes liées au XML

Je sépare dans cet objet ce qui est lié à la manipulation des XML car un certain nombre d'explications est nécessaire. L'écriture d'un fichier XML est assez simple avec les méthodes intégrées dans ADO.NET mais le fichier XML obtenu peut être sensiblement différent selon le paramétrage des objets.

Nous allons utiliser le code suivant comme base de travail pour l'étude des propriétés XML de l'objet Dataset

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
string strSQL = "SELECT * FROM Authors";
OleDbDataAdapter MonDA = new OleDbDataAdapter(strSQL, strConn);
DataSet MonDataset = new DataSet("Auteurs");
//MonDataset.Namespace = "DemoName"
//MonDataset.Prefix = "DemoPrefix"
MonDataset.WriteXmlSchema("c:\\tutoriel\\schema.xml");
}
```

Propriétés NameSpace et Prefix

Permettent de définir un espace de nom et un préfixe XML lors de l'écriture du fichier. Si cela n'a pas d'importance pour l'écriture du fichier, cela peut permettre de limiter les informations récupérées lors de la lecture. En effet, un Dataset ne récupère que les données ayant le même espace de nom et le même préfixe que lui lors de la lecture d'un fichier XML.

Méthode WriteXMLSchéma & ReadXmlSchema

Dans le cas de notre exemple, le fichier résultant sera de la forme :

```
<?xml version="1.0" standalone="yes" ?>
- <xs:schema id="Auteurs" xmlns=""
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  - <xs:element name="Auteurs" msdata:IsDataSet="true" msdata:Locale="fr-
      FR">
    - <xs:complexType>
      - <xs:choice maxOccurs="unbounded">
        - <xs:element name="Table">
          - <xs:complexType>
            - <xs:sequence>
                <xs:element name="Au_ID" type="xs:int"
                  minOccurs="0" />
                <xs:element name="Author" type="xs:string"
                  minOccurs="0" />
                <xs:element name="Year_x0020_Born"
                  type="xs:short" minOccurs="0" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:schema>
```

Je vous ai parlé plus avant des objets Dataset fortement typés, c'est à dire construits autour d'une source définie. Nous pouvons nous demander si notre schéma XML est identique au schéma d'un Dataset fortement typé de la même table.

```
<?xml version="1.0" standalone="yes"?>
<xs:schema id="AuteurTyped"
targetNamespace="http://www.tempuri.org/AuteurTyped.xsd"
xmlns:mstns="http://www.tempuri.org/AuteurTyped.xsd"
xmlns="http://www.tempuri.org/AuteurTyped.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-
com:xml-msdata" attributeFormDefault="qualified" elementFormDefault="qualified">
  <xs:element name="AuteurTyped" msdata:IsDataSet="true" msdata:Locale="fr-FR">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Authors">
          <xs:complexType>
            <xs:sequence>
             <xs:element name="Au_ID" msdata:AutoIncrement="true" type="xs:int" />
              <xs:element name="Author" type="xs:string" minOccurs="0" />
              <xs:element name="Year_x0020_Born" type="xs:short" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
    <xs:unique name="Constraint1" msdata:PrimaryKey="true">
      <xs:selector xpath=".//mstns:Authors" />
      <xs:field xpath="mstns:Au_ID" />
    </xs:unique>
</xs:element>
</xs:schema>
```

Nous voyons que tel n'est pas le cas. Un Dataset fortement typé possède des informations de schéma que mon schéma ne possède pas. Je pourrais pourtant aisément reproduire le même fichier en récupérant correctement le schéma de ma source et en faisant attention au nom lors du mappage des données. Le schéma obtenu par la méthode WriteXMLSchema produit donc bien un fichier de Schéma XSD, mais celui-ci sera composé des informations de schéma présentes dans le Dataset.

La méthode ReadXmlSchema permet de donner un schéma au Dataset par la lecture d'un fichier XML, celui-ci pouvant éventuellement contenir des données.

Méthode InferXMLSchéma.

Cette méthode doit être utilisée avec précaution voire ignorée en raison des risques qu'elle présente. Cette méthode tente de définir le schéma en fonction des données du fichier XML. Vous pouvez être à peu près certains que tous vos champs seront de types chaîne et qu'il n'y aura aucune contrainte.

Méthode WriteXML & ReadXml

Pour ces méthodes comme pour les précédentes je travaille à partir d'un fichier. Cependant elles acceptent aussi des objets TextReader, TextWriter, XMLReader et XMLWriter ainsi que les objets Stream.

La méthode WriteXML écrit un Dataset avec les données qu'il contient. Nous verrons dans l'étude des DataColumn et des DataRelation qu'il peut être écrit différemment. La méthode accepte un paramètre important XMLWriteMode qui peut prendre les valeurs suivantes :

Nom	Description
DiffGram	Permet d'écrire les modifications en cours dans le fichier XML.
IgnoreSchema	Écrit le contenu actuel de DataSet en tant que données XML.
WriteSchema	Écrit le contenu actuel de DataSet en tant que données XML, avec la structure relationnelle comme schéma XSD.

La méthode ReadXML remplit un Dataset en partant d'un fichier XML. A l'identique de la méthode WriteXML, elle accepte un paramètre XMLReadMode qui peut prendre une des valeurs ci-dessous :

Nom	Description
Auto	Membre par défaut. Tente d'utiliser le mode le plus approprié
DiffGram	Lit un DiffGram en affectant les modifications du DiffGram au DataSet. La sémantique est identique à celle d'une opération Merge. Comme lors de l'opération Merge, les valeurs RowState sont conservées. L'entrée pour ReadXml à l'aide de DiffGrams doit uniquement être obtenue en utilisant la sortie de WriteXml en tant que DiffGram. Le DataSet cible doit avoir le même schéma que le DataSet sur lequel WriteXml est appelé en tant DiffGram. Sinon, l'opération de fusion du DiffGram échoue, provoquant la levée d'une exception.
Fragment	Lit des documents XML, tels ceux qui sont générés suite à l'exécution de requêtes FOR XML, sur une instance de SQL Server. Lorsque la valeur de XmlReadMode est Fragment, l'espace de noms par défaut est lu en tant que schéma inline.
IgnoreSchema	Ignore tout schéma inline et lit les données dans le schéma DataSet existant. Si des données ne correspondent pas au schéma existant, elles sont ignorées (y compris les données d'espaces de noms différents définis pour le DataSet). Si les données sont un DiffGram, IgnoreSchema fonctionne comme DiffGram.
InferSchema	Ignore tout schéma inline, obtient le schéma à partir des données et charge ces dernières. Si DataSet contient déjà un schéma, le schéma en cours est étendu par l'ajout de nouveaux tableaux ou l'ajout de colonnes aux tableaux existants. Une exception est levée si le tableau obtenu existe déjà avec un espace de noms différent, ou si une ou plusieurs des colonnes obtenues entrent en conflit avec des colonnes existantes.
ReadSchema	Lit tout schéma inline et charge les données. Si DataSet contient déjà un schéma, de nouveaux tableaux peuvent être ajoutés à ce dernier, mais une exception est levée si un tableau du schéma inline existe déjà dans DataSet.

Nous reviendrons dans la troisième partie sur la manipulation des fichiers XML.

Evènements

MergeFailed

Se déclenche lors de l'échec de la méthode Merge c'est à dire lorsque les schémas ne peuvent être fusionnés en fonction du paramètre MissingSchemaAction.

La classe DataTable

Cet objet est très similaire au Dataset et conceptuellement identique à une table de base de donnée. On peut voir la DataTable comme le croisement d'une collection de colonnes (les champs) et de lignes (les données). La table est donc container d'une collection de DataColumn et de DataRow. Elle possède aussi des contraintes. Dès à présent il faut faire la différence entre les contraintes d'une table et celles d'une colonne. N'hésitez pas à lire la documentation du site de Frédéric Brouard.

La plupart des propriétés et méthodes sont les mêmes que pour l'objet Dataset je n'y reviendrais pas ici.

Les noms des tables ne sont normalement pas sensibles à la casse, sauf si deux tables existent avec une casse différente dans le même Dataset. Faite attention lors de la création de vos tables.

Construction d'une table par le code

Lorsqu'on souhaite ajouter une table par programmation, on doit respecter un ordre bien défini.

- Ajout des colonnes
- Ajout de la table au Dataset
- Ajout des contraintes
- Ajout des lignes

Il est tout à fait de possible de remplir une table créée par le code avec des données tirées d'une source existantes. En effet, il n'y a pas récupération du schéma sur un schéma existant si ceux-ci sont identiques. L'ajout des lignes quant à lui repose toujours sur une question de clé primaire.

Mise à jour des tables

Quel que soit le mode de remplissage de la table (Merge d'un Dataset, lecture XML, Ajout par code…) il y a toujours comparaison des valeurs de clé primaire si celle-ci existe. Si la clé primaire n'existe pas toute opération de ligne sur la table ajoute une ligne. Si la clé primaire existe, il y a mise à jour d'une ligne possédant une valeur de clé identique à une ligne existante et ajout sinon.

Je vais prendre un exemple simple pour illustrer mon propos.

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
string strSQL = "SELECT * FROM Authors";
OleDbDataAdapter MonDA = new OleDbDataAdapter(strSQL, strConn);
DataSet MonDataset = new DataSet("Auteurs");
//MonDA.FillSchema(MonDataset, SchemaType.Source)
MonDA.Fill(MonDataset);
MessageBox.Show(MonDataset.Tables[0].Rows.Count.ToString());
MonDA.Fill(MonDataset);
MessageBox.Show(MonDataset.Tables[0].Rows.Count.ToString());

}
```

En l'état mon code va afficher 6246 puis 12492. Par contre si je valide la ligne en commentaire, il va afficher deux fois 6246. Ceci vient du fait que la ligne de récupération de schéma va attribuer la contrainte de clé primaire à la colonne 'Au_Id'. Dès lors, comme toutes les lignes existent déjà dans la table, il y aura rafraîchissement des lignes. Ce code est un exemple. N'oubliez pas que les récupérations de schéma sont très onéreuses en terme d'efficacité.

Méthodes de DataTableCollection

Add

Ajoute une table à la collection des tables du Dataset. La méthode autorise le passage d'une chaîne ou d'un objet DataTable comme paramètre. Comme je vous l'ai signalé, il vaut mieux créer le schéma de la table avant de l'ajouter à la collection.

```
MonDA.Fill(MonDataset);
DataTable MaTable =new DataTable("Correcteurs");
MaTable.Columns.Add("Co_id", Type.GetType("System.Int32")).AutoIncrement
= true;
MaTable.Columns.Add("Co_Name", Type.GetType("System.String"));
MaTable.Constraints.Add("pk1", MaTable.Columns[0], true);
MonDataset.Tables.Add(MaTable);
```

 Attention au respect de la casse dans la syntaxe "System.Int32"

AddRange

Ajoute un tableau d'objets DataTable à la fin de la collection.

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
string strSQL = "SELECT * FROM Authors";
OleDbDataAdapter MonDA = new OleDbDataAdapter(strSQL, strConn);
DataSet MonDataset = new DataSet("Auteurs");
//MonDA.FillSchema(MonDataset, SchemaType.Source)
MonDA.Fill(MonDataset);
DataTable MaTable1 =new DataTable("Correcteurs");
MaTable1.Columns.Add("Co_id", Type.GetType("System.Int32")).AutoIncrement
= true;
MaTable1.Columns.Add("Co_Name", Type.GetType("System.String"));
MaTable1.Constraints.Add("pk1", MaTable1.Columns[0], true);
DataTable MaTable2 =new DataTable("Correcteurs_Auteurs");
MaTable2.Columns.Add("Co_id", Type.GetType("System.Int32"));
MaTable2.Columns.Add("Au_id", Type.GetType("System.Int32"));
MaTable2.PrimaryKey = new DataColumn[] {MaTable2.Columns[0],
MaTable2.Columns[1]};
MonDataset.Tables.AddRange(new DataTable[] {MaTable1, MaTable2});
MessageBox.Show(MonDataset.Tables.Count.ToString());
}
```

CanRemove

Vérifie si une table peut être supprimée de la collection.

Clear

Supprime toutes les tables de la collection.

Contains

Renvoie vrai si la table dont le nom est passé en paramètre appartient à la collection.

IndexOf

Renvoie la position ordinale de la table dont le nom est passé en paramètre.

Remove & RemoveAt

Supprime la table passée en paramètre. Accepte le nom ou l'objet. RemoveAt fonctionne avec l'index.

```
if (MonDataset.Tables.Contains("Correcteurs"))
   if (MonDataset.Tables.CanRemove(MonDataset.Tables["Correcteurs"]))
    MonDataset.Tables.RemoveAt(MonDataset.Tables.IndexOf("Correcteurs"));
```


La syntaxe de cet exemple est très mauvaise. Il ne s'agit ici que d'exemple de possibilités d'écriture.

Propriétés de DataTable

ChildRelations (DataRelationCollection)

Renvoie la collection des relations enfants de l'objet DataTable. Les relations enfants sont celles ou la table cible fournit sa clé primaire comme clé étrangère de la table enfant.

Columns (DataColumnCollection)

Renvoie la collection des objets DataColumn de la table.

Constraints (ConstraintCollection)

Renvoie la collection des contraintes de la table.

DefaultView (DataView)

Renvoie le DataView associé à la table.

ParentRelations (DataRelationCollection)

Renvoie la collection des relations parents de l'objet DataTable. Les relations parents sont celles ou la table cible contient la clé étrangère.

PrimaryKey (DataColumn)

Définit ou renvoie un tableau d'objets DataColumn qui contient la ou les colonnes composant la clé primaire. Il s'agit d'une alternative à l'utilisation de la collection des contraintes.

Rows (DataRowCollection)

Renvoie la collection des lignes de la table.

Méthodes de DataTable

BeginLoadData & EndLoadData

Désactive ou active les notifications, la gestion d'index et les contraintes lors du chargement de données par la méthode LoadDataRow.

Compute

Public Object Compute(String expression**, String** filter**);**
Calcule la valeur passée dans expression sur les lignes qui respectent le paramètre Filter.

```
Object objMoy;
objMoy = MonDataset.Tables[0].Compute("avg([year born])", "[year born] is
not null");
textBox1.Text = Convert.ToString(objMoy);
```

ImportRow

Importe la ligne passée en paramètre dans la Table. La ligne est importée telle quelle et conserve sa valeur d'état.

LoadDataRow

Public DataRow LoadDataRow(Object [] values**, Boolean** *fAcceptChanges***);**

Modifie ou ajoute une ligne avec les valeurs passées dans le tableau. Le tableau values() doit contenir autant de valeurs qu'il y a de champs dans la table. Si un champ possède une valeur par défaut vous pouvez passer la valeur NULL pour que la ligne prenne la valeur par défaut. Passez aussi NULL dans les colonnes auto-incrémentées pour que la table génère la valeur.

La méthode utilise le tableau values() pour calculer la valeur de la clé primaire dans le contexte de la table. Si une clé équivalente existe dans la table, la ligne est mise à jour sinon elle est ajoutée.

Si fAcceptChanges vaut True ou est omis, la méthode AcceptChanges est appelée sur la ligne, sinon la valeur d'état de la ligne passe à Added ou Modified.

Cette méthode s'utilise généralement avec les méthodes BeginLoadData et EndLoadData.

NewRow

Crée une nouvelle ligne qui respecte le schéma de la table. Cette ligne n'est pas implicitement ajoutée à la table, vous devez donc gérer cet ajout par votre code. A la différence de la méthode précédente, il doit y avoir respect des contraintes pour la ligne ajoutée. Regardons l'exemple suivant.

```
MonDA.Fill(MonDataset);
Object[] RowValues  = {1, "Rabilloud J-M", 1967};
MonDataset.Tables[0].PrimaryKey = new
DataColumn[]{MonDataset.Tables[0].Columns[0]};
MonDataset.Tables[0].BeginLoadData();
MonDataset.Tables[0].LoadDataRow(RowValues, true);
MonDataset.Tables[0].EndLoadData();
DataRow MaLigne = MonDataset.Tables[0].NewRow();
MaLigne[MonDataset.Tables[0].Columns[0]] = 1;
MaLigne[MonDataset.Tables[0].Columns[1]] = "Rabilloud J-M";
MaLigne[MonDataset.Tables[0].Columns[2]] = 1967;
MonDataset.Tables[0].Rows.Add(MaLigne);
```

La méthode LoadDataRow va remplacer une ligne existante, mais vous aurez une exception lors de l'ajout de la ligne 'MaLigne' puisque la ligne existe déjà.

Select

Permet de récupérer un tableau d'objet DataRow répondant à un filtre. Le tableau peut être éventuellement trié.

Public DataRow[] Select(String *filterExpression***, String** *sort***, DataViewRowState** *recordStates***);**

Cette syntaxe est une surcharge, il est possible d'omettre des paramètres. Examinons ceux-ci successivement. N'oubliez pas que l'objet renvoyé est toujours un tableau d'objets DataRow.

FilterExpression est l'équivalent d'une clause WHERE SQL sans le mot clé WHERE. Pour ceux d'entre vous qui avez utilisé la méthode Find ADO, la syntaxe est identique.

Sort est l'équivalent de la clause ORDER BY SQL sans le mot clé ORDER BY. On utilise le mot clé DESC pour trier dans l'ordre décroissant.

RecordStates permet de limiter la méthode à des lignes possédant des états particuliers.

```
MonDA.Fill(MonDataset);
DataTable MaTable = MonDataset.Tables[0];
DataRow[] tabResult;
tabResult = MaTable.Select("Au_id<100");
MessageBox.Show(tabResult.GetUpperBound(0).ToString());
tabResult = null;
tabResult = MaTable.Select("Au_id<100", "Author DESC");
MessageBox.Show(Convert.ToString( tabResult[0][0]));
tabResult = null;
tabResult = MaTable.Select("Au_id<100", " Author DESC",
DataViewRowState.Added);
MessageBox.Show(Convert.ToString(tabResult.GetUpperBound(0)));
```

Il est possible d'utiliser des caractères génériques pour le filtre et de composer les critères. Un exemple valide est :

```
tabResult = MaTable.Select("Author like '%john%' AND [year born] IS NOT NULL");
```

N'oubliez pas que la définition de la propriété CaseSensitive peut influer sur le tri.

Les règles sur les délimiteurs s'appliquent.

Evènements de DataTable

ColumnChanging & ColumnChanged

```
    void MaTable_ColumnChanged(Object sender,
System.Data.DataColumnChangeEventArgs e)
    void MaTable_ColumnChanging(Object sender,
System.Data.DataColumnChangeEventArgs e)
```

Se produit chaque fois que la valeur d'une colonne est modifiée. Ce concept est toujours un peu difficile à cerner car il faut voir dans ce cas la colonne non comme le champ stricto sensu mais plutôt comme le champ d'une ligne. Il s'agit donc bien des modifications de données.

Les arguments de l'événement renvoient la ligne et la colonne ainsi que la valeur proposée pour la donnée.

RowChanging & RowChanged

```
    void MaTable_RowChanging(Object sender,System.Data.DataRowChangeEventArgs e)
    void MaTable_RowChanged(Object sender, System.Data.DataRowChangeEventArgs e)
```

Se produit lorsqu'une ligne est modifiée. Les arguments de l'événement renvoient la ligne concernée ainsi qu'un paramètre action qui défini l'action en cause. Celui-ci peut prendre une des valeurs suivantes.

Nom	Description	Valeur
Nothing	La ligne n'a pas été modifiée.	0
Delete	La ligne a été supprimée de la table.	1
Change	La ligne a été modifiée.	2
Rollback	La dernière modification de la ligne a été annulée.	4
Commit	Les modifications de la ligne ont été validées.	8
Add	La ligne a été ajoutée à la table.	16

RowDeleting & RowDeleted

Se produit lorsqu'une ligne est supprimée de la table.

La classe DataColumn

Un objet DataColumn représente le champ d'une table. Il n'est jamais possible d'accéder à une valeur de données uniquement par le champ. Il y a donc une dualité dans le concept de colonne. Soit-on entend par-là, la colonne d'une table qui représente un champ, soit la colonne d'une ligne qui cible une valeur. Dans ce chapitre, c'est l'objet 'champ' qui nous intéresse.

Colonne auto-incrémentée

S'il est vrai que l'enfer est pavé de bonnes intentions, nous tenons là une très belle dalle. Ce problème n'est d'ailleurs pas tant dû aux colonnes 'auto-incrémentées' qu'aux colonnes de clés primaires non informationnelles. Je développe brièvement. Une école de pensée des SGBD consiste à dire qu'une bonne clé est une clé purement informatique, sur une seule colonne de la taille du mot du processeur ; c'est dans ce contexte que s'utilise une colonne auto-incrémentée. Si je ne conteste pas qu'en termes de vitesse ce soit idéal, dans une approche multi utilisateurs je suis beaucoup plus réservé. Une colonne auto-incrémentée revient à demander au SGBD d'attribuer une valeur forcément unique à chaque enregistrement. Comme elle n'est liée en aucune manière à une quelconque information, elle ne garantit pas l'absence de doublons et elle ne permet pas d'avoir le moindre commencement de soupçons de la présence d'un enregistrement identique. Bref il s'agit d'un mécanisme qui utilisé seul est assez inefficace. C'est pour cela qu'en général on ajoute une contrainte d'unicité sur un échantillon de colonne de la table.

Mais revenons à nos moutons, votre table dans votre Dataset n'est pas liée à la source de données. Lorsque vous définissez une colonne auto-incrémentée dans votre DataTable, vous allez devoir lui attribuer une valeur d'incrément et une valeur de départ, en vous assurant qu'elle ne risque pas de donner une valeur existant déjà dans la table. Une idée qui vient spontanément consiste à prendre comme valeur de départ la plus grande valeur présente dans la table. Il s'agit pourtant d'une des plus mauvaises idées dans la longue histoire des mauvaises idées. En effet, il existe une bonne chance que le numéro que le SGBD fournira entre dans une sorte de conflit masqué avec vos propres numéros. Je vous donne un exemple de ce scénario catastrophe dans la troisième partie.

Sachez dès à présent que les colonnes auto-incrémentées d'une base de données vous forceront à une programmation plus ou moins immonde pour gérer correctement le cas.

Type de données

Une colonne de table possède toujours un type. Le type de votre table doit être cohérent avec celui de la table source. Il y a un mappage entre les types de votre table et ceux de la table source.

Utiliser les types de votre espace de nom

Chaque espace de nom possède une énumération (OleDBType par exemple) qui permet de faire l'équivalence entre le type de la source et le type du DataColumn. Lorsque vous créez votre table par le code dans l'intention d'y rapatrier des données, vous devez chercher un type acceptable sans quoi il y aura levée d'une exception s'il y a déperdition d'information.

Utiliser des types SQL-Server

Ceci n'est possible que pour un travail avec SQL-Server. Il existe un ensemble de classes contenues dans l'espace de nom 'System.Data.SqlTypes'. Il ne faut pas confondre cela avec les énumérations de type présents dans les espaces de noms SQL, OLEDB et ODBC. Le tableau suivant montre le mappage des types.

Type SQL-Server natif	Type SqlTypes .NET Framework	Type SqlDbType .NET Framework
binary	SqlBinary	Binary
Bigint	SqlInt64	BigInt
Char	SqlString	Char
datetime	SqlDateTime	DateTime
decimal	SqlDecimal	Decimal
Float	SqlDouble	Float
image	SqlBinary	Image
Int	SqlInt32	Int
Money	SqlMoney	Money
nchar	SqlString	NChar

Type SQL-Server natif	Type SqlTypes .NET Framework	Type SqlDbType .NET Framework
Ntext	SqlString	NText
nvarchar	SqlString	NVarChar
Numeric	SqlDecimal	Numeric
Real	SqlSingle	Real
smalldatetime	SqlDateTime	SmallDateTime
smallint	SqlInt16	SmallInt
smallmoney	SqlMoney	SmallMoney
sql_variant	Object	Variant
sysname	SqlString	VarChar
text	SqlString	Text
timestamp	SqlBinary	TimeStamp
tinyint	SqlByte	TinyInt
varbinary	SqlBinary	VarBinary
varchar	SqlString	VarChar
uniqueidentifier	SqlGuid	UniqueId

Il y a plusieurs avantages inhérents à l'utilisation des classes SqlTypes.

- La rapidité : Pour communiquer entre les types du FrameWork et les types natifs, il y a conversion dans ces classes. Travailler directement avec diminue le temps de conversion.
- La sécurité : Il y a une nette diminution du risque de faute de conversion.
- Un certain nombre de fonctionnalités sont accessibles avec ses classes. Il y a notamment la levée d'exception en cas de troncature ou la gestion pour tous les types de la valeur Null.

Méthode Add de DataColumnCollection

Il existe plusieurs versions surchargées. Utilisez de préférence
Public void Add(DataColumn *column***);**
Plutôt que
Public virtual DataColumn Add(string columnName, Type type);

Propriétés de DataColumn

AllowDbNull (Booléen)

Définit ou renvoie True si la colonne accepte les valeurs NULL

AutoIncrement, AutoIncrementSeed & AutoIncrementStep

AutoIncrement attend un booléen. S'il est vrai, ADO.NET génèrera des valeurs pour les lignes ajoutées avec les valeurs définies dans AutoIncrementSeed et AutoIncrementStep (int32). Si vous tentez de convertir une colonne existante en colonne AutoIncrement, son type sera converti en Int32. Ceci n'est pas possible si la colonne est une colonne contenant une expression.

Nous verrons dans la troisième partie comment gérer leurs utilisations.

```
MonDA.Fill(MonDataset);
DataTable MaTable  = MonDataset.Tables[0];
MaTable.Columns[0].AutoIncrement = True;
MaTable.Columns[0].AutoIncrementSeed = -1;
MaTable.Columns[0].AutoIncrementStep = -1;
```

Caption (String)

Définit la légende de la colonne. Si elle n'est pas définie, c'est la propriété ColumName qui sera renvoyée.

ColumnMapping (MappingType)

Obtient ou définit le MappingType de la colonne. Peut prendre une des valeurs suivantes :

Nom	Description
Attribute	La colonne est mappée à un attribut XML.
Element	La colonne est mappée à un élément XML.
Hidden	La colonne est mappée à une structure interne.
SimpleContent	La colonne est mappée à un nœud XmlText.

Cette propriété définit comment le fichier XML sera écrit lors de l'appel de la méthode WriteXML.
Si je reprends mon exemple de schéma XML, avec le code :

```
MonDA.Fill(MonDataset);
foreach (DataColumn MaColonne in MonDataset.Tables[0].Columns)
     MaColonne.ColumnMapping = MappingType.Attribute;
MonDataset.WriteXmlSchema("c:\\tutoriel\\schema2.xml");
```

J'obtiendrais

```
<?xml version="1.0" standalone="yes" ?>
- <xs:schema id="Auteurs" xmlns=""
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  - <xs:element name="Auteurs" msdata:IsDataSet="true" msdata:Locale="fr-
      FR">
    - <xs:complexType>
      - <xs:choice maxOccurs="unbounded">
        - <xs:element name="Table">
          - <xs:complexType>
              <xs:attribute name="Au_ID" type="xs:int" />
              <xs:attribute name="Author" type="xs:string" />
                  <xs:attribute name="Year_x0020_Born"
                type="xs:short" />
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
  </xs:element>
</xs:schema>
```

ColumnName (String)

Définit ou renvoie le nom de la colonne. Utilisez impérativement des noms sans espaces si vos colonnes servent dans des expressions. De manière générale, utilisez des noms sans espaces.

DataType (Type)

Représente le type de la colonne. Ce type doit être un des types gérés par le Framework, indiqués ci-dessous.

- Boolean
- Byte
- Char
- DateTime
- Decimal
- Double
- Int16
- Int32
- Int64
- SByte
- Single
- String
- TimeSpan
- UInt16
- UInt32
- UInt64

Une exception est générée si vous modifiez cette propriété alors que la colonne a commencé à stocker des données.

DefaultValue (Object)

Définit la valeur par défaut d'une colonne. Cette valeur est automatiquement ajoutée dans les lignes nouvellement créées. La valeur doit être définie après le type et doit évidemment respecter celui-ci.

Expression (String)

Cette propriété permet de remplacer élégamment le DataShape Provider. Elle permet de faire des calculs dans une colonne, soit utilisant une ou plusieurs colonnes de la table, soit d'écrire des fonctions d'agrégation.

Les exemples suivants sont réalisés avec la base NorthWind SQL-Server.

Calcul d'une expression dépendant d'une autre colonne :

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "packet size=4096;user id=sa;data source=NOM-
SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=northwind;password=monpasse";
string strSQL = "SELECT ProductID, UnitPrice, UnitsInStock FROM
Products";
SqlDataAdapter MonDa = new SqlDataAdapter(strSQL, strConn);
DataSet MonDataSet = new  DataSet();
MonDa.Fill(MonDataSet);
MonDataSet.Tables[0].Columns.Add("Taxe",
System.Type.GetType("System.Decimal"), "UnitPrice*0.196");
dataGrid1.DataSource = MonDataSet.Tables[0].DefaultView;
}
```

Calcul d'un agrégat

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "packet size=4096;user id=sa;data source=NOM-
SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=northwind;password=monpasse";
string strSQL = "SELECT ProductID, UnitPrice, UnitsInStock FROM
Products";
SqlDataAdapter MonDa = new SqlDataAdapter(strSQL, strConn);
DataSet MonDataSet = new  DataSet();
MonDa.Fill(MonDataSet);
MonDataSet.Tables[0].Columns.Add("Compte",
System.Type.GetType("System.Decimal"), "Avg(UnitPrice)");
dataGrid1.DataSource = MonDataSet.Tables[0].DefaultView;
}
```

Dans ce genre de calcul c'est beaucoup moins efficace puisque la valeur est reproduite sur toutes les lignes.

Simulation d'un filtre

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "packet size=4096;user id=sa;data source=NOM-
SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=northwind;password=monpasse";
string strSQL = "SELECT ProductID, UnitPrice, UnitsInStock FROM
Products";
SqlDataAdapter MonDa = new SqlDataAdapter(strSQL, strConn);
DataSet MonDataSet = new  DataSet();
MonDa.Fill(MonDataSet);
MonDataSet.Tables[0].Columns.Add("Filtre",
System.Type.GetType("System.String"), "iif(UnitPrice>50,'vrai','faux')");
dataGrid1.DataSource = MonDataSet.Tables[0].DefaultView;
}
```

MaxLength (Integer)

Définit un nombre maximum de caractères pour une colonne de type texte. Renvoie –1 si elle n'est pas définie.

Ordinal (Integer)

Renvoie la position ordinale de la colonne. Renvoie –1 si la colonne n'est pas dans une table.

ReadOnly (Booléen)

Renvoie ou définit si les valeurs du champ peuvent être modifiées.

Unique (Booléen)

Renvoie ou définit l'unicité des lignes de la colonne.

La classe DataRow

Les objets DataRow contiennent les données de la table. La manipulation des lignes d'une table est toujours liée à un travail sur les données. Cette collection est centrale dans les applications impliquant des mises à jour de la source de données. Ce sont les objets DataRow qui récupèrent les informations d'erreurs et de modification.

Les valeurs de la ligne

Une ligne contient plusieurs valeurs pour représenter ses données. Ce concept est familier au développeurs ADO. Une ligne peut posséder une ou plusieurs des valeurs suivantes

DataRowVersion	Description
Current	Les valeurs actuelles de la ligne. Cette version de ligne n'existe pas pour les lignes dont le RowState est Deleted.
Default	Version de ligne par défaut d'une ligne particulière. La version de ligne par défaut d'une ligne Added, Modified ou Unchanged est Current. La version de ligne par défaut d'une ligne Deleted est Original. La version de ligne par défaut d'une ligne Detached est Proposed.
Original	Les valeurs d'origine de la ligne. Cette version de ligne n'existe pas pour les lignes ayant un RowState Added.
Proposed	Les valeurs proposées de la ligne. Cette version de ligne existe pendant une opération de modification sur une ligne ou pour une ligne qui ne fait pas partie d'un DataRowCollection.

Il est possible de savoir si une version particulière existe en utilisant la méthode HasVersion. En général vous n'avez pas besoin de lire les différentes versions d'une ligne. Par contre celles-ci sont indispensables à l'écriture des requêtes de mises à jour.

L'état d'une ligne

Les lignes possèdent toujours un état. Celui-ci permet de savoir si la ligne possède des modifications en attente ou non. L'état peut prendre une des valeurs suivantes.

RowState	Description
Unchanged	Aucune modification n'a été apportée depuis le dernier appel à AcceptChanges ou depuis la création de la ligne par DataAdapter.Fill.
Added	La ligne a été ajoutée à la table mais la méthode AcceptChanges n'a pas été appelée.
Modified	Certains éléments de la ligne ont été modifiés.
Deleted	La ligne a été supprimée d'une table et AcceptChanges n'a pas été appelé.
Detached	Detached est défini pour une ligne qui a été créée mais qui ne fait pas partie d'un DataRowCollection. Le RowState d'une ligne créée prend la valeur Detached. Une fois le DataRow ajouté au DataRowCollection à l'aide de l'appel à la méthode Add, la propriété RowState prend la valeur Added. Detached est également défini pour une ligne qui a été supprimée d'un DataRowCollection à l'aide de la méthode Remove ou par la méthode Delete suivie de la méthode AcceptChanges.

Vous pouvez utiliser l'état pour filtrer les lignes d'une table. C'est aussi par l'état que le DataAdapter détermine la commande à appliquer.

Remontée des erreurs

Lorsqu'une ligne ne peut transmettre ses modifications vers la source, il y a écriture d'une chaîne décrivant l'erreur dans la propriété RowError de la ligne. Cette propriété permet de traiter les erreurs a posteriori de la mise à jour. Il ne faut pas perdre de vue que seule une chaîne est stockée. Dans certains cas, la lecture et l'interprétation de l'erreur ne sont pas aisées. La ligne n'a pas nécessairement une erreur dans sa propriété RowError même si une erreur est survenue. Il faut parfois utiliser les méthodes GetColumnError et GetColumnsInError pour trouver une erreur spécifique à un champ de la ligne.

Ajout et modification de ligne

On peut être un peu perturbé par la profusion de techniques permettant d'ajouter une ligne. De manière générale, on utilise NewRow puis un Add sur la collection. On utilise ImportRow lors du transfert de la ligne d'une table vers une autre. Enfin on se sert de LoadDataRow lorsqu'on souhaite qu'ADO.NET ajoute ou mette à jour une ligne éventuellement existante.

La modification des lignes suit globalement deux principes.

- On sélectionne une ligne puis on modifie ses valeurs soit directement soit en mode édition
- On ajoute une ligne en comptant sur la clé primaire pour assurer les modifications.

Evidemment la première méthode est beaucoup plus propre.

Une autre méthode est la modification indirecte. Lorsqu'on manipule des objets DataRow dans un tableau, celui-ci fais référence aux objets de la table. Une modification dans le tableau engendre une modification dans la table.

```
DataRow[] TabLigne = MonDataset.Tables(0).Select("Au_Id < 10");
TabLigne[0][1] = "Mézigue";
dataGrid1.DataSource = MonDataset.Tables[0];
```

Méthode de DataRowCollection

En outre des méthodes habituelles des collections ADO.NET, cette collection possède deux méthodes spécifiques Find et InsertAt.

Add

Permet l'ajout d'une ligne à la collection, soit en passant un objet DataRow, soit avec un tableau de valeurs.

Find

Permet une recherche d'une ligne par la valeur de sa clé primaire. Il n'est pas possible de spécifier un champ pour la recherche. Si la clé primaire est composée de plusieurs champs, vous devez passer un tableau de valeurs à la méthode

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
string strSQL = "SELECT * FROM Authors";
OleDbDataAdapter MonDA = new OleDbDataAdapter(strSQL, strConn);
DataSet MonDataset = new DataSet("Auteurs");
MonDA.Fill(MonDataset);
MonDataset.Tables[0].Constraints.Add("pk1",
MonDataset.Tables[0].Columns[0], true);
DataRow MaLigne = MonDataset.Tables[0].Rows.Find("1");
MessageBox.Show(Convert.ToString(MaLigne[1]));
strSQL = "SELECT * FROM [Title Author]";
MonDA.SelectCommand.CommandText = strSQL;
DataTable Matable =new DataTable();
MonDA.Fill(Matable);
Matable.Constraints.Add("pk", new DataColumn[] {Matable.Columns[0],
Matable.Columns[1]}, true);
MonDataset.Tables.Add(Matable);
MaLigne = MonDataset.Tables[1].Rows.Find(new Object[] {"0-0133656-1-4",
"1454"});
if (MaLigne != null)
     MessageBox.Show("Reussie");
        }
```

InsertAt

Permet l'insertion à une position spécifique dans la collection.

void InsertAt(DataRow *row***, Integer** *pos***)**

Attention, cette méthode ne fonctionne pas pour un tableau d'objet DataRow mais uniquement pour un objet DataRowCollection. N'oubliez pas non plus que la notion de position de ligne n'a pas de sens pour un SGBD. Celle-ci n'a d'importance que pour l'écriture d'un fichier XML ou pour la navigation dans votre collection.

Propriétés de l'objet DataRow

HasErrors (Booléen)

Indique si la ligne contient un message d'erreur. Cette propriété est en lecture seule mais vous pouvez la modifier en jouant sur la propriété RowError.

Item (Object)

Surchargée. Permet de récupérer la valeur d'un champ de la ligne en spécifiant éventuellement la version.

public object this [DataColumn column, DataRowVersion version] {get;}

Le paramètre column peut être l'objet DataColumn, la position ordinale ou le nom de la colonne. Pour des raisons de performance, n'utilisez pas le nom de la colonne. Si vous ne précisez pas la version c'est la valeur courante qui est renvoyée.

ItemArray (Object())

Renvoie ou définit un tableau d'objet contenant les valeurs de tous les champs de la ligne. Cette méthode ne possède pas de surcharge permettant de spécifier la version. Lors de l'utilisation de cette méthode pour modifier une ligne, passez des paramètres Nothing pour obtenir les valeurs AutoIncrement ou par défaut d'une colonne.

RowError (String)

Définit ou obtient le texte d'un message d'erreur affecté à la ligne.

RowState (DataRowState)

Lecture seule. Obtient l'état de la ligne spécifiée. Voici un petit exemple de manipulation de ligne.

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
string strSQL = "SELECT * FROM Authors";
OleDbDataAdapter MonDA = new OleDbDataAdapter(strSQL, strConn);
DataSet MonDataset = new DataSet("Auteurs");
MonDA.Fill(MonDataset);
DataTable MaTable  = MonDataset.Tables[0];
MaTable.Columns[0].AutoIncrement = true;
MaTable.Columns[0].AutoIncrementSeed = -1;
MaTable.Columns[0].AutoIncrementStep = -1;
MaTable.Constraints.Add("pk1", MonDataset.Tables[0].Columns[0], true);
MaTable.BeginLoadData();
MaTable.LoadDataRow(new Object[] {1, "Rabilloud", 1967}, false);
MaTable.EndLoadData();
DataRow maLigne = MaTable.NewRow();
maLigne.ItemArray = new Object[] {null, "Mezigue", 1930};
MaTable.Rows.Add(maLigne);
maLigne = MaTable.Rows.Find(10);
maLigne.Delete();
DataRow[] TabLigne = MaTable.Select("", "", DataViewRowState.Added |
DataViewRowState.Deleted | DataViewRowState.ModifiedCurrent);
MessageBox.Show(Convert.ToString(TabLigne.GetLength(0)));
}
```

Méthode de DataRow

AcceptChanges

Valide les modifications en attente sur la ligne. Si la propriété RowState de la ligne était Added ou Modified, elle devient Unchanged. Si la propriété RowState est Deleted, la ligne est supprimée. Fait aussi un appel implicite de la méthode EndEdit. Attention cela n'induit pas un quelconque envoi des modifications vers la source de données. Prenons le cas suivant

```
MaTable.Constraints.Add("pk1", MonDataset.Tables[0].Columns[0], true);
MaTable.BeginLoadData();
MaTable.LoadDataRow(new Object[] {1, "Rabilloud", 1967}, false);
MaTable.EndLoadData();
DataRow maLigne = MaTable.NewRow();
maLigne.ItemArray = new Object[] {null, "Mezigue", 1930};
MaTable.Rows.Add(maLigne);
MaTable.AcceptChanges();
MessageBox.Show(maLigne.RowState.ToString());
```

Dans ce cas, lors de l'appel de la mise à jour, la ligne quoique récemment ajoutée, sera marquée comme étant inchangée. Dès lors elle ne sera pas envoyée vers la source de données.

BeginEdit, CancelEdit & EndEdit

Ces méthodes permettent de procéder à la modification des lignes en mode édition. Les contrôles dépendants utilisent toujours ce principe. L'utilisation du mode édition n'est jamais imposée mais elle permet de suspendre les opérations de validation et les événements de changement jusqu'à l'appel de la méthode EndEdit.

```
MonDA.Fill(MonDataset);
DataTable MaTable  = MonDataset.Tables[0];
DataRow maligne = MaTable.Rows[0];
maligne.BeginEdit();
maligne[1] = textBox1.Text;
if (Len(maligne[1].DataRowVersion.Proposed) > 0)
     maligne.EndEdit();
else
     maligne.CancelEdit();
```

ClearErrors

Efface toutes les erreurs de la ligne qu'elles soient ou non affectées à un champ.

Delete

Marque une ligne comme 'Deleted'. Si l'état de la ligne était 'Added' la ligne est définitivement supprimée. Une ligne marquée 'Deleted' peut être restaurée par l'appel de la méthode RejectChanges.

GetChildRows & GetParentRows

Permet la récupération des lignes dans un tableau de DataRow au travers d'une relation spécifiée. Accepte un paramètre pour préciser la version de la ligne désirée.

Public DataRow[] GetChildRows(DataRelation *relation***, DataRowVersion** *version***) ;**

Le paramètre relation peut être aussi défini par son nom.

GetColumnError

Public String GetColumnError(DataColumn colum);

Permet de récupérer l'erreur d'une colonne spécifiée de la ligne. Le paramètre colonne peut être aussi passé comme nom ou comme position ordinale.

GetColumnsInError

Renvoie un tableau de DataColumn contenant les colonnes en erreurs de la ligne. Regardons l'exemple suivant de gestion d'erreurs.

```
DataTable MaTable  = MonDataset.Tables[0];
DataRow maLigne = MaTable.Rows[0];
if (maLigne.HasErrors)
     {
     DataColumn[] TabErrCol= maLigne.GetColumnsInError();
     foreach(DataColumn MaColonne in TabErrCol)
MessageBox.Show(Convert.ToString(maLigne.GetColumnError(MaColonne)));
     maLigne.ClearErrors();
     }
```

HasVersion

Public Boolean HasVersion(DataRowVersion *version***);**

Permet de définir si la version passée en paramètre existe pour la ligne. S'utilise généralement en conjonction avec les méthodes GetChildRows et GetParentRows ou avant l'appel de la propriété Item.

IsNull

Public Boolean IsNull(DataColumn *column***, DataRowVersion** *version***);**
Le paramètre colonne peut aussi être la position ordinale ou le nom mais dans ce cas la version ne peut être précisée. Renvoie vrai si la valeur de la colonne est NULL.

RejectChanges

Annule les modifications en attente. Si RowState est 'Deleted' ou 'Modified', les valeurs précédentes de la ligne sont rétablies et RowState devient 'Unchanged'. Si RowState est 'Added', la ligne est supprimée. Appelle aussi CancelEdit implicitement.

SetColumnError

Public void SetColumnError(DataColumn *column***, String** *error***);**
Affecte à la colonne définie en paramètre le texte d'erreur contenu dans la chaîne error.

SetParentRow

Public void SetParentRow(DataRow *parentRow***, DataRelation** *relation***);**
Définit la nouvelle ligne parent de la ligne. Vous n'êtes pas obligé de spécifier la Relation mais je vous conseille vivement de le faire.

La Classe Constraint

La collection Constraints représente les contraintes d'une table. Les contraintes de colonne sont des propriétés de l'objet DataColumn. Il y a là un piège sémantique important. Une colonne peut posséder la contrainte unique par sa propriété Unique et être la cible d'une contrainte UniqueConstraint de la table. Lorsque je dis qu'il s'agit d'une contrainte de table, je commets une approximation. La seule véritable contrainte de table est la clé primaire, c'est à dire une contrainte 'UniqueConstraint' particulière. Les autres ont pour cible une colonne ou un groupe de colonnes.
Cette collection ne contient que deux types possibles 'UniqueConstraint' et 'ForeignKeyConstraint'.

UniqueConstraint

Contrainte d'unicité verticale ou horizontale. Lorsqu'il s'agit d'une colonne ceci revient à mettre la propriété unique de la colonne à Vrai. L'ajout d'une colonne ayant la propriété unique à une table provoque l'ajout d'une contrainte de ce type à la collection ConstraintCollection de la table.
La clé primaire de la table est une contrainte 'UniqueConstraint' de la collection.
La méthode Add de la collection permet d'ajouter des contraintes d'unicité. Soit vous créez un objet Constraint que vous passez à la méthode Add, soit vous pouvez utiliser la version surchargée suivante.
Public virtual Constraint Add(string *name***, DataColumn** *column***, bool** *primaryKey***);**
Où name est le nom de la contrainte, column l'objet DataColumn ciblé et primaryKey un booléen définissant si cette contrainte est la clé primaire.
Nous avons vu de nombreux exemples de création de contrainte, je n'en réécris pas ici.

Propriétés de UniqueConstraint

Columns (DataColumn())

Renvoie le tableau des objets DataColumn contenus dans la contrainte

IsPrimaryKey (Booléen)

Renvoie vrai si la contrainte est la clé primaire de la table.

ForeignKeyConstraint

La contrainte de clé étrangère, appelée aussi contrainte d'intégrité référentielle ne se conçoit que dans le cadre d'une relation. En fait une clé étrangère est composée d'une ou plusieurs colonnes, celles-ci étant la clé primaire de la table mise en relation. L'ajout d'une relation induit l'ajout d'une 'ForeignKeyConstraint' à la collection. Nous verrons cela dans l'étude des DataRelation. Voici un exemple de manipulation de clé étrangère et d'une relation.

```
private void button1_Click(object sender, System.EventArgs e)
{
string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
string strSQL = "SELECT * FROM Publishers";
OleDbDataAdapter MonDAPublisher = new OleDbDataAdapter(strSQL, strConn);
DataSet MonDataset = new DataSet("Pubs");
MonDAPublisher.Fill(MonDataset, "Editeurs");
DataTable maTable = MonDataset.Tables[0];
maTable.Constraints.Add("pk1", maTable.Columns[0], true);
strSQL = "SELECT * FROM Titles";
OleDbDataAdapter MonDATitle = new OleDbDataAdapter(strSQL, strConn);
MonDATitle.Fill(MonDataset, "Titres");
maTable = MonDataset.Tables[1];
maTable.Constraints.Add("pk1", maTable.Columns["ISBN"], true);
DataRelation DRelat = new DataRelation("EditeurTitre",
MonDataset.Tables[0].Columns[0], maTable.Columns["PubId"], true);
MonDataset.Relations.Add(DRelat);
ForeignKeyConstraint FKCont = (ForeignKeyConstraint)
maTable.Constraints[1];
FKCont.AcceptRejectRule = AcceptRejectRule.Cascade;
FKCont.DeleteRule = Rule.Cascade;
FKCont.UpdateRule = Rule.Cascade;
maTable = MonDataset.Tables[0];
maTable.Columns.Add("NbLivre", Type.GetType("System.Int32"),
"Count(child(EditeurTitre).ISBN)");
dataGrid1.DataSource = maTable;
        }
```

Propriétés de ForeignKeyConstraint

AcceptRejectRule

Définit les règles à appliquer en cas d'appel de la méthode AcceptChanges. Peut prendre la valeur :

- None : Aucune action ne se produit
- Cascade : Les actions se répercutent en cascade

Les actions sont définies dans les propriétés DeleteRule et UpdateRule

Columns (DataColumn())

Renvoie le tableau des objets DataColumn contenus dans la contrainte

DeleteRule & UpdateRule(Rule)

Renvoie ou définit la répercussion de l'action lorsqu'une ligne parent est supprimée ou modifiée. Peut prendre une des valeurs suivantes :

Nom	Description
Cascade	Supprime ou modifie les lignes enfants. Il s'agit de la valeur par défaut.
None	Aucune action n'est effectuée sur les lignes enfants.
SetDefault	Affecte la valeur contenue dans la propriété **DefaultValue** comme valeur des lignes enfants.
SetNull	Affecte DBNull comme valeur des lignes enfants.

Attention, mettre ces propriétés à None n'empêche pas la levée d'exceptions.

RelatedColumns (DataColumn())

Renvoie un tableau d'objet DataColumn représentant les colonnes parents de la relation.

RelatedTable (DataTable)

Renvoie la table parente de la relation

La classe DataRelation

Les objets DataRelation appartiennent à l'objet Dataset. Une relation permet de mettre en correspondance les lignes d'une ou de plusieurs tables. Elle se base sur l'utilisation d'une clé étrangère qui permet d'identifier une ligne 'parente'. La mise en relation des informations permet de ne pas dupliquer les données et donc d'alléger considérablement le SGBD. Dans ADO.NET vous pouvez créer des relations entres des objets d'un même Dataset. On distingue quatre types de relation:

- **Relation réflexive** : Celle-ci n'utilise qu'une seule table. La clé étrangère permet de faire référence à une ligne de la même table, donc de créer une vision 'hiérarchique' de la table.

```
string strConn, strSQL;
strConn = @"packet size=4096;user id=sa;data source= NOM-
SZ71L7KTH17\\TUTO ;persist security info=False;initial
catalog=northwind;password=monpasse"
strSQL = "SELECT * FROM Employees";
SqlClient.SqlDataAdapter MonDa = new SqlClient.SqlDataAdapter(strSQL,
strConn);
DataSet MonDataSet = new DataSet();
MonDa.Fill(MonDataSet);
Data.DataTable maTable = MonDataSet.Tables(0);
maTable.Constraints.Add("pk1", maTable.Columns(0), true);
MonDataSet.Relations.Add("AutoReflex", maTable.Columns(0),
maTable.Columns("ReportsTo"), true);
maTable.Columns.Add("NbSubord", Type.GetType("System.Int32"),
"Count(child(AutoReflex).ReportsTo)");
this.DataGrid1.DataSource = maTable;
```

- **Relation un à un** : Une telle relation associe une ligne enfant avec une ligne parent. Dans ce seul cas la clé étrangère doit être unique. Les relations de ce type sont très rares car il est alors plus simple de mettre tous les champs dans une seule table.
- **Relation un à plusieurs** : C'est le type le plus courant. Une ligne parent peut avoir plusieurs lignes enfants, mais une ligne enfant ne peut avoir qu'un seul parent. Celle-ci s'obtient par la recopie d'une ou plusieurs colonnes ayant la contrainte d'unicité dans la table enfants.
- **Relation plusieurs à plusieurs** : Stricto sensu il ne s'agit plus d'une relation, mais de deux relations utilisant une table de jointure. Une table de jointure comporte les colonnes composant la clé primaire des deux tables, sa propre clé primaire étant définie sur l'ensemble des colonnes.

Les relations présentent aussi l'avantage des performances. Il est souvent moins lourd de créer deux tables puis une relation que de créer une table utilisant une requête avec jointure.

Sachez aussi qu'il est possible de désigner des lignes enfants dans l'expression d'une colonne calculée.

Remarque de Maxence Hubiche

Je ne suis pas d'accord avec cette assertion *« Stricto sensu, il ne s'agit pas d'une relation mais de deux relations utilisant une table de jointure »*. Sur le plan de la modélisation, il s'agit d'une association entre 2 entités. Il est vrai qu'une telle association nécessitera, au moment de la création de la base de données, la génération d'une table intermédiaire qui est en fait le résultat de l'association entre les 2 entités du modèle. Mais là, nous sommes dans le modèle conceptuel. Par contre, dans la base de données (modèle physique) le cas pourra être légèrement différent : Imagine que tu aies à lier (pour je ne sais quelle raison) 2 champs quelconques (avec doublons), de 2 tables quelconques. Il est plus que probable que tu auras une vraie relation n-n, puisque chaque valeur de la table 1 sera susceptible d'être associée à n valeurs de la table 2, et vice-versa.

Réponse : Je ne saisie pas pourquoi une relation n:n avec des doublons serait plus une vraie relation n:n que s'il n'y a pas de doublons. Sur le plan conceptuel tu as raison, mais je n'aborde pas le plan conceptuel dans le cadre de ce cours, je décris comment il faudra gérer les objets DataRelation dans ADO.NET.

Réponse de M. Hubiche

C'est le 'Stricto Sensu' qui m'ennuie, parce que tout dépend de l'endroit où l'on se situe :

Sur le plan analytique, la relation N-N existe. Qu'il y ait des doublons ou pas !

Maintenant, il est vrai que sur le plan physique, on ne peut la représenter autrement que par 2 relations 1-N. Qu'il y ait des doublons ou pas ...

Remarque de Maxence Hubiche

Puis-je te poser une question ? (Histoire de vraiment t'ennuyer☺)
Imagine la table des matériels.
Avec ID, Description, Prix.
Imagine que, maintenant, tu veuilles connaître la quantité de matériels composant d'autres matériels, de telle sorte que tu puisses récupérer les compositions. Cette fois, l'association est une association plusieurs-à-plusieurs sur la même table…
Comment tu gères ?
Pourtant, c'est une relation réflexive aussi ☺
Je t'énerve là ?

Réponse : Une légère contrariété tout au plus ☹ . Je ne suis pas bien sur de voir là ou tu veux en venir. Mais dans le cas que tu cites, il s'agit d'une relation plusieurs à plusieurs. La notion de relation de type n:n n'induit pas l'existence de deux tables. Il s'agit bien d'une relation entre des enregistrements. Tu auras donc une table de jointure faisant référence dans ses champs à la clé primaire de la même table. C'est certes une relation reflexive, mais elle nécessite plus d'une table.

Réponse de M. Hubiche

Ce qui m'a ennuyé c'est que tu limites, dans ta définition, la relation réflexive à une relation 1-N. ALors que dans l'exemple que je te donne, il s'agit aussi d'une relation N-N, et elle est aussi réflexive.

En fait, la réflexivité porte sur le fait que l'association se fait d'une entité sur elle-même. Les cardinalités interviennent peu ici.

La collection DataRelationCollection

Elle se manipule avec les mêmes méthodes que DataTableCollection.

Méthode Add

Surchargée. Comme d'habitude vous pouvez soit créer une relation puis l'ajouter à la collection, soit utiliser une des syntaxes surchargées de création directe.

public DataRelation Add(String *name***, DataColumn***[] parentColumns***, DataColumn[]** *childColumns***, bool** *createConstraints***)**

Le nom est facultatif.

CreateConstraint permet de créer implicitement la contrainte de clé étrangère sur la table enfant.

ParentColumn et ChildColumn peuvent être soit un tableau de DataColumn, soit un seul DataColumn.

Propriétés de DataRelation

ChildColumns (DataColumn())

Renvoie un tableau d'objet DataColumn (ou un seul DataColumn) composant la clé étrangère de la table enfant.

ChildKeyConstraint (ForeignKeyConstraint)

Renvoie l'objet ForeignKeyConstraint de la relation.

ChildTable (DataTable)

Renvoie la table enfant.

ParentColumns, ParentTable, ParentKeyConstraint

Fonctionne comme les propriétés précédentes pour les objets parents. Dans le cas de ParentKeyConstraint c'est un objet UniqueConstraint qui est renvoyé.

Nested (Booléen)

Renvoie vrai si les objets DataRelation sont imbriqués. Cette propriété est utilisée lorsqu'on écrit un fichier XML ou pour synchroniser un XMLDataDocument avec le Dataset.

```
<CustomerOrders>
  <Customers>
    <CustomerID>ALFKI</CustomerID>
    <CompanyName>Alfreds Futterkiste</CompanyName>
  </Customers>
  <Customers>
    <CustomerID>ANATR</CustomerID>
    <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
  </Customers>
  <Orders>
    <OrderID>10643</OrderID>
    <CustomerID>ALFKI</CustomerID>
    <OrderDate>1997-08-25T00:00:00</OrderDate>
  </Orders>
  <Orders>
    <OrderID>10692</OrderID>
    <CustomerID>ALFKI</CustomerID>
    <OrderDate>1997-10-03T00:00:00</OrderDate>
  </Orders>
  <Orders>
    <OrderID>10308</OrderID>
    <CustomerID>ANATR</CustomerID>
    <OrderDate>1996-09-18T00:00:00</OrderDate>
  </Orders>
</CustomerOrders>
```

Notez que l'élément Customers et les éléments Orders sont représentés en tant qu'éléments frères. Si vous souhaitiez que les éléments Orders apparaissent comme les enfants de leurs éléments parents respectifs, il vous faudrait assigner la valeur true à la propriété Nested du DataRelation.

Le code suivant montre la sortie que vous obtiendriez, avec les éléments Orders imbriqués dans leurs éléments parents respectifs.

```
<CustomerOrders>
  <Customers>
    <CustomerID>ALFKI</CustomerID>
    <Orders>
      <OrderID>10643</OrderID>
      <CustomerID>ALFKI</CustomerID>
      <OrderDate>1997-08-25T00:00:00</OrderDate>
    </Orders>
    <Orders>
      <OrderID>10692</OrderID>
      <CustomerID>ALFKI</CustomerID>
      <OrderDate>1997-10-03T00:00:00</OrderDate>
    </Orders>
    <CompanyName>Alfreds Futterkiste</CompanyName>
  </Customers>
  <Customers>
    <CustomerID>ANATR</CustomerID>
    <Orders>
      <OrderID>10308</OrderID>
      <CustomerID>ANATR</CustomerID>
      <OrderDate>1996-09-18T00:00:00</OrderDate>
    </Orders>
    <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
  </Customers>
</CustomerOrders>
```

Les classes DataView & DataViewManager

Jusque là, je ne vous ai pas tellement parlé de navigation dans les données ni de contrôles dépendants. Beaucoup de développeurs ADO sont pourtant perturbés par la disparition des méthodes de navigation de l'objet Recordset. Si ces méthodes avaient un sens dans un recordset qui gère une position courante, elle n'en a plus dans un Dataset qui est un ensemble de collections. En effet, la ligne en cours d'une collection n'est jamais qu'une position ordinale et naviguer dans une collection ne revient qu'à modifier celle-ci.

Les objets DataView sont une couche entre les objets DataTable et l'interface utilisateur. Elles permettent de présenter la table sous-jacente comme on veut. Une table peut exposer plusieurs vues d'elle-même. Un DataView est lié dynamiquement à la table, ce qui veut dire qu'il reflète les données et les changements apportés à la table.

Il ne faut pas confondre un DataView et une vue au sens SQL du terme. Un DataView ne peut exposer que des données provenant d'une seule table, ne peut pas créer de colonne de calcul et ne peut pas restreindre les colonnes affichées.

Un DataViewManager permet de gérer les paramètres des vues de toutes les tables au niveau du Dataset. En lui-même il ne fait rien de bien particulier si ce n'est permettre d'afficher des données connexes dans des contrôles dépendants.

Je vous parlais un peu plus avant de navigation dans le jeu d'enregistrement.

La navigation au sens ADO du terme se gère par l'intermédiaire du DataBinding. Non parce que ce n'est pas possible de faire autrement pour naviguer dans les enregistrements, mais parce que c'est la méthode pour pouvoir lier à la conception des contrôles dépendants gérant la navigation. La navigation dans une collection ne pose aucun problème. Regardons le code suivant :

```
private int monIndex;
    this.Form1.Load += new System.EventHandler(Form1_Load);
    this.Btn_Del.Click += new System.EventHandler(Btn_Del_Click);
    this.cmdFirst.Click += new System.EventHandler(cmdFirst_Click);
    this.cmdNext.Click += new System.EventHandler(cmdNext_Click);
    this.cmdPrev.Click += new System.EventHandler(cmdPrev_Click);
    this.cmdLast.Click += new System.EventHandler(cmdLast_Click);
    …
    private void Form1_Load(object sender, System.EventArgs e) {
        string strConn, strSQL;
        strConn = @"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
        strSQL = "SELECT * FROM Authors";
        Data.OleDb.OleDbDataAdapter MonDAPublisher = new
Data.OleDb.OleDbDataAdapter(strSQL, strConn);
        DataSet MonDataset = new DataSet("Pubs");
        MonDAPublisher.Fill(MonDataset, "Auteurs");
        MaTable = MonDataset.Tables(0);
        maTable.Constraints.Add("pk1", maTable.Columns(0), true);
        monIndex = 0;
        Affiche_Donnee();
    }

    private void Btn_Del_Click(object sender, System.EventArgs e) {
        MaTable.Rows.RemoveAt(monIndex);
    }
    private void cmdFirst_Click(object sender, System.EventArgs e){
        monIndex = 0;
        Affiche_Donnee();
    }
    private void Affiche_Donnee() {
        MaLigne = MaTable.DefaultView.Item(monIndex);
        this.TextBox1.Text = MaLigne.Item(0);
        this.TextBox2.Text = MaLigne.Item(1);
        this.TextBox3.Text = MaLigne.Item(2) & "";
    }
    private void cmdNext_Click(object sender, System.EventArgs e){
        monIndex++;
        Affiche_Donnee();
    }
    private void cmdPrev_Click(object sender, System.EventArgs e){
        monIndex-;
        Affiche_Donnee();
    }
    private void cmdLast_Click(object sender, System.EventArgs e) {
        monIndex = MaTable.DefaultView.Count – 1;
        Affiche_Donnee();
    }
```

Vous verrez que le code fonctionne très bien même si je retire des lignes de la table. Cela vient du fait que le DataView gère toujours correctement ces index.

Propriétés de DataView

AllowDelete, AllowEdit & AllowNew (Booléen)

Obtient ou définit si la modification de table spécifiée est autorisée au travers du DataView.

ApplyDefaultSort (Booléen)

Applique le tri par la clé primaire comme tri par défaut. Il faut évidemment que la propriété Sort soit vide et qu'il y ait une clé primaire.

Count (Integer)

Renvoie le nombre d'enregistrements visibles après l'application des filtres.

RowFilter (String)

Utilise une chaîne de filtre, de manière identique à un filtre sur une expression de colonne, c'est à dire une clause WHERE SQL sans le mot clé WHERE. Les chaînes suivantes sont des chaînes valides :

"Au_Id > 100"
"Author like '%John%'"
"Isnull([year born],'Null Column') = 'Null Column'"
"BirthDate =< #01/01/2004#"
Et toutes les combinaisons avec les opérateurs logiques.

RowStateFilter (DataRowState)

Renvoie ou définit un filtre en fonction de l'état et/ou de la version de la ligne. Il est possible de combiner plusieurs valeurs.

Sort (String)

Renvoie ou définit une chaîne de tri, identique à la clause 'ORDER BY'. Les colonnes sont triées par ordre d'apparition dans la chaîne ; on utilise DESC pour un tri en ordre décroissant.

Méthodes du DataView

AddNew

Ajoute une ligne à un DataView.

Find

Identique à la méthode du DataRow.

DataRowView

Représente une ligne d'un objet DataView. Ces lignes se manipulent sensiblement comme des DataRow mais possèdent des propriétés supplémentaires.

Propriétés

IsEdit (Booléen)

Renvoie vrai si le DataRowView est en mode édition.

IsNew (Booléen)

Renvoie vrai si le DataRowView a été ajouté.

Item (Object)

Renvoie ou définit la valeur dans la colonne passée en paramètre comme nom ou position ordinale.

Row (DataRow)

Renvoie l'objet DataRow désigné par le DataRowView.

RowVersion (DataRowVersion)

Renvoie la version de la ligne qui est utilisée pour l'affichage. (originale, courante, proposée)

Manipulation des DataView

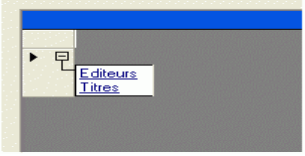
Les DataView se manipulent comme des DataTable, mais en ajoutant des fonctionnalités de présentation. Vous n'avez pas nécessairement besoin de code pour utiliser les DataView puisqu'il existe un composant dans l'onglet 'Données' de la palette.

Lorsqu'on manipule les DataView par le code, c'est toujours dans le cadre d'une liaison avec des contrôles. Sinon ils ne font rien que vous ne pourrez faire avec un Select sur des DataRow. Nous verrons dans la troisième partie des exemples d'utilisation.

DataViewManager

Cet objet par contre n'est quasiment jamais utilisé, car il est rare de vouloir gérer des vues de données non connexes. Toutefois, il est possible de lier un DataViewManager au contrôle DataGrid. Dans ce cas, celui-ci vous affichera une liste de table que vous pourrez ajouter. Si je reprends mon exemple

```
this.Button1.Click += new System.EventHandler(Button1_Click);
    …
    private void Button1_Click(object sender, System.EventArgs e) {
        string strConn, strSQL;
        strConn = @"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
        strSQL = "SELECT * FROM Publishers";
        OleDbDataAdapter MonDAPublisher = new OleDbDataAdapter(strSQL,
strConn);
        DataSet MonDataset = new DataSet("Pubs");
        MonDAPublisher.Fill(MonDataset, "Editeurs");
        Data.DataTable maTable = MonDataset.Tables(0);
        maTable.Constraints.Add("pk1", maTable.Columns(0), true);
        strSQL = "SELECT * FROM Titles";
        OleDbDataAdapter MonDATitle = new OleDbDataAdapter(strSQL,
strConn);
        MonDATitle.Fill(MonDataset, "Titres");
        maTable = MonDataset.Tables(1);
        maTable.Constraints.Add("pk1", maTable.Columns("ISBN"), true);
        DataRelation DRelat = new DataRelation("EditeurTitre",
MonDataset.Tables(0).Columns(0), maTable.Columns("PubId"), true);
        MonDataset.Relations.Add(DRelat);
        ForeignKeyConstraint FKCont = maTable.Constraints.Item(1);
        FKCont.AcceptRejectRule = AcceptRejectRule.Cascade;
        FKCont.DeleteRule = Rule.Cascade;
        FKCont.UpdateRule = Rule.Cascade;
        maTable = MonDataset.Tables(0) ;
        maTable.Columns.Add("NbLivre", Type.GetType("System.Int32"),
"Count(child(EditeurTitre).ISBN)") ;
        this.DataGrid1.DataSource = MonDataset.DefaultViewManager;
    }
```

J'obtiendrais une grille comme cela. 

Les fournisseurs managés

Nous allons maintenant regarder les classes composant les fournisseurs managés. A la fin de cette étude nous verrons comment fonctionne un Dataset lié à un DataAdapter pour une communication Bidirectionnelle.

Comme précédemment nous allons beaucoup travailler sur le fournisseur OLEDB ; néanmoins vous trouverez plusieurs exemples utilisant le fournisseur Managé pour SQL-Server chaque fois que celui-ci engendrera des différences de codage.

Je vais maintenant expliciter quelques termes que j'emploie dans la suite de ce document.

Lot de requêtes

Certains SGBD supportent le principe du lot de requêtes. Ceci consiste à envoyer plusieurs requêtes par le code ou par le biais d'une procédure stockée et à récupérer les informations en un seul lot. Les lots de requêtes sont gérés par ADO.NET mais ils comportent un certain nombre de pièges qu'il faut connaître.

Procédures stockées & Requêtes stockées

Une requête stockée, parfois improprement appelée vue est une requête dont la définition se situe dans le SGBD. Ce peut être indifféremment une requête action, sélection ou de manipulation de structure. Celle-ci peut attendre des paramètres d'entrée.

Une procédure stockée est un ensemble d'ordre SQL, écrits dans un langage propre au SGBD. Elle peut combiner plusieurs types de requêtes et peut utiliser des paramètres d'entrée, de sortie et de retour.

Que ce soit des requêtes ou des paramètres, il appartient au développeur de savoir ce qu'elles attendent et/ou renvoient.

La connexion

Comme dans ADO, l'objet OledbConnection représente une session unique d'accès à la source de données. La connexion gère les exceptions et la sécurité de l'accès.

Regroupement de connexions

Ce mécanisme appelé 'connection pooling' est toujours implémenté dans ADO.NET comme il l'est dans ADO. Les fournisseurs proposent un jeu de connexions disponible pour augmenter les performances. Lorsqu'on demande la fermeture d'une connexion, celle-ci retourne dans le pool toujours ouverte et ne se fermera effectivement qu'au dépassement de son paramètre LifeTime. Ce principe, méconnu de la plupart des développeurs permet l'utilisation de connexions dites 'implicites' avec des performances acceptables.

Le pool peut être configuré par le biais de la chaîne de connexion à l'aide des paramètres suivants :

Nom	Valeur par défaut	Description
Connection Lifetime	0	Lorsqu'une connexion est retournée au pool, l'heure de sa création est comparée à l'heure actuelle et la connexion est détruite si cet intervalle de temps (en secondes) excède la valeur spécifiée par Connection Lifetime. Ceci est utile dans les configurations en clusters pour forcer l'équilibrage de la charge entre un serveur en cours d'exécution et un serveur qui vient d'être mis en ligne. La valeur zéro (0) aura pour conséquence un délai d'attente maximal pour les connexions regroupées.
Connection Reset	True	Détermine si la connexion de base de données est réinitialisée lorsqu'elle est supprimée du pool. Pour Microsoft SQL Server version 7.0, False évite un aller-retour supplémentaire du serveur lors de l'établissement d'une connexion, mais vous devez savoir que l'état de connexion, comme le contexte de base de données, n'est pas réinitialisé.
Enlist	True	Si la valeur est True, le dispositif de regroupement inscrit automatiquement la connexion dans le contexte de transaction en cours du thread de création si un contexte de transaction existe.
Max Pool Size	100	Nombre maximal de connexions autorisées dans le pool.
Min Pool Size	0	Nombre minimal de connexions conservées dans le pool.
Pooling	True	Si la valeur est True, la connexion est retirée du pool approprié ou, si nécessaire, créée et ajoutée au pool approprié.

Sauf cas particulier on ne définit que le nombre maximum de connexions dans le pool.

Il ne faut pas perdre de vue qu'un pool ne contient que des connexions créées par des chaînes de connexions strictement identiques. Une petite modification de cette dernière sous-entend la création d'un nouveau pool. Le pool se subdivise seul lorsqu'une transaction est définie, il n'y a donc pas de risque de récupérer une connexion ayant un contexte de transaction.

La mise en pool des connexions se justifie pleinement dans les applications n-tiers où les objets métier libèrent et reprennent fréquemment des connexions. Elle est assez inopérante dans les applications Client/serveur.

Pour empêcher la mise en pool des connexions il suffit d'ajouter "OLEDB Services = -4;" dans votre chaîne de connexion pour le fournisseur OLE DB ou "Pooling=False" pour le fournisseur SQL-Server

Propriétés

ConnectionString (String)

C'est la définition de la chaîne de connexion. A la différence d'ADO, on ne peut plus morceler la construction de la chaîne de connexion. Celle-ci peut avoir une écriture assez complexe lorsqu'on gère la sécurité. Vous trouverez dans l'aide la liste des paramètres utilisables. Rappelons-nous que la définition de cette propriété n'ouvre pas la connexion. Une chaîne non valide lève une exception ArgumentException. Il n'est pas possible de modifier la chaîne d'une connexion ouverte. La plupart de ces paramètres se retrouvent dans les propriétés ci-dessous, mais elles sont maintenant en lecture seule.

```
string strConn = @"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;Mode=Share Deny None;Jet OLEDB:Engine
Type=5;Jet OLEDB:Database Locking Mode=1;Jet OLEDB:Global Partial Bulk
Ops=2;Jet OLEDB:Global Bulk Transactions=1;Jet OLEDB:Create System
Database=False;Jet OLEDB:Encrypt Database=False;Jet OLEDB:Don't Copy
Locale on Compact=False;Jet OLEDB:Compact Without Replica
Repair=False;Jet OLEDB:SFP=False" ;
        Data.OleDb.OleDbConnection MaConn = new
Data.OleDb.OleDbConnection(strConn);
        MaConn.Open() ;
```

Cas du fournisseur SQL

Celui-ci attend une chaîne de connexion ne possédant pas de paramètre Provider. N'utilisez pas une chaîne utilisant le fournisseur SqlOledb pour le fournisseur managé SQL.

```
strConn = @"workstation id=NOM-SZ71L7KTH17;packet size=4096;user
id=sa;data source=NOM-SZ71L7KTH17\\TUTO;persist security
info=False;initial catalog=northwind;password=monpasse";
        Data.SqlClient.SqlConnection MaConn = new
Data.SqlClient.SqlConnection(strConn);
        MaConn.Open() ;
```

ConnectionTimeOut (Integer)

Vaut 0 si le Timeout n'est pas défini, sinon représente la durée en seconde (15 par défaut).

Database & DataSource (String)

Renvoie le nom de la source et le nom du serveur.

PacketSize (Integer)

Spécifique au fournisseur managé pour SQL-Server.

Représente la taille (octets) des paquets transitant par le réseau. Si vous devez transmettre des données de gros volume, il est souvent recommandable d'augmenter la taille. Attention cette modification doit être faite dans la chaîne de connexion.

State (ConnectionState)

Renvoie l'état de la connexion.

Nom	Valeur	Description
Closed	0	La connexion est fermée.
Open	1	La connexion est ouvertee.
Connecting	2	La connexion est en cours
Executing	4	Une commande est en train de s'exécuter
Fetching	8	Des données sont en cours de transfert
Broken	16	La connexion a été interrompue par une cause différente de close

Les quatre dernières valeurs ne sont pas encore utilisées dans C#.

Méthodes

BeginTransaction

Dans ADO.NET seule existe la méthode BeginTransaction sur l'objet Connection. Dans ADO.NET on utilise un objet Transaction crée par l'appel de BeginTrans. Ceci présente une bien meilleure lisibilité des transactions. La syntaxe est :

public override SqlTransaction **BeginTransaction(**IsolationLevel *iso***, string** *transactionName***)**

Les deux paramètres sont facultatifs. TransactionName permet de nommer la transaction ; isolationLevel peut prendre une des valeurs suivantes :

Nom	Valeur	Description
Unspecified	-1	Le fournisseur utilise un niveau d'isolation différent de celui qui est spécifié mais ce niveau ne peut pas être déterminé.
Chaos	16	Vous ne pouvez pas écraser les changements en attente des transactions dont le niveau d'isolation est supérieur.
ReadUncommitted	256	À partir d'une transaction, vous pouvez voir les modifications des autres transactions non encore engagées.
ReadCommitted	4096	Valeur utilisée par défaut. A partir d'une transaction, vous pouvez afficher les modifications d'autres transactions uniquement lorsqu'elles ont été engagées.
adXactRepeatableRead	65536	À partir d'une transaction, vous ne pouvez pas voir les modifications effectuées dans d'autres transactions, mais cette nouvelle requête peut renvoyer de nouveaux jeux d'enregistrements.
Serializable	1048576	Les transactions sont conduites en isolation d'autres transactions.

Vous noterez qu'à l'exception des noms, c'est exactement le même tableau que dans ADO.

Pour plus de renseignement sur les transactions, reportez-vous à la section 'objet Transaction' dans ce chapitre.

ChangeDatabase

Cette propriété permet à la connexion ciblée de changer de base de données avec le **même** fournisseur. Ceci présente un intérêt pour remplir un Dataset avec des données d'origines variée. Il faut toutefois faire très attention lors du changement de base car il peut y avoir génération de nombreuses exceptions.

Close

Ferme la connexion.

CreateCommand

Renvoie un objet Command lié à la connexion. Cela revient à faire de façon explicite l'approche que l'on pouvait avoir avec la méthode Execute dans ADO. La différence majeure vient du fait que la commande ainsi crée n'est pas définie. On doit la manipuler comme un objet Command classique.

Open

Ouvre la connexion

Evènements

La gestion des évènements est un peu différente que celle dont vous aviez l'habitude avec VB

Rappels sur les évènements

Les événements en C# se programment en utilisant des délégués.
Un exemple classique est

```
this.Button1.Click += new System.EventHandler(Button1_Click);
    …
    private void Button1_Click(object sender, System.EventArgs e) {
        string strConn = @"Provider=Microsoft.Jet.OLEDB.4.0;User
ID=Admin;Data Source=C:\\tutoriel\\biblio.mdb;Mode=Share Deny None;Jet
OLEDB:Engine Type=5;Jet OLEDB:Database Locking Mode=1;Jet OLEDB:Global
Partial Bulk Ops=2;Jet OLEDB:Global Bulk Transactions=1;Jet OLEDB:Create
System Database=False;Jet OLEDB:Encrypt Database=False;Jet OLEDB:Don't
Copy Locale on Compact=False;Jet OLEDB:Compact Without Replica
Repair=False;Jet OLEDB:SFP=False";
        MaConn.ConnectionString = strConn;
        MaConn.Open();
        MaConn.InfoMessage += new
OleDb.OleDbInfoMessageEventHandler(MaConn_InfoMessage);
    }

    private static void MaConn_InfoMessage(object sender,
Data.OleDb.OleDbInfoMessageEventArgs args) {
        foreach(Data.OleDb.OleDbError err in args.Errors) {
            ……
        }
    }
```

Dans ce cas, il vous faut ensuite gérer ce qui revient dans EventArgs et écrire le code de gestion (dans notre exemple : parcours de la collection des erreurs)

C'est dans la programmation évènementielle que la connexion a le plus changé. En effet il n'y a plus que deux évènements pour la connexion

InfoMessage

C'est de loin mon événement préféré. Cela fait longtemps que j'utilise ADO et je peux dire que cet événement n'aura jamais fait ce que je pouvais espérer. En théorie cela permet de récupérer les messages 'sans gravité'. Comme l'interprétation du 'sans gravité' varie selon le fournisseur, c'est une programmation heureuse qui vous attend.

Dans la théorie, une erreur 'pas grave' déclenche un événement InfoMessage, une erreur 'grave' lève une exception et une erreur 'très grave' ferme la connexion. Nous voilà bien avancés.

Pour les aventuriers, regardons le code suivant qui devrait être utilisé pour la gestion des erreurs du fournisseur.

```
this.Form1.Load += new System.EventHandler(Form1_Load);
    …
    private void Form1_Load(object sender, System.EventArgs e) {
        cn.ConnectionString = @"workstation id=NOM-SZ71L7KTH17;packet
size=4096;user id=sa;data source="NOM-SZ71L7KTH17\\TUTO";persist security
info=False;initial catalog=northwind;password=monpasse";
        try {
            cn.Open();
            cn.StateChange += new StateChangeEventHandler(cn_StateChange);
            cn.InfoMessage += new
SqlInfoMessageEventHandler(cn_InfoMessage1);
        catch(Exception ex) {
            MessageBox.Show("là c'est grave", "ERREUR",
MessageBoxButtons.OK, MessageBoxIcon.Error) ;
        }
    }

    private void cn_InfoMessage1(object sender,
System.Data.SqlClient.SqlInfoMessageEventArgs e) {
        MessageBox.Show("Là c'est moins grave", "ERREUR",
MessageBoxButtons.OK, MessageBoxIcon.Error);
    }

    private void cn_StateChange(object sender,
System.Data.StateChangeEventArgs e) {
        if(e.CurrentState == ConnectionState.Closed && e.OriginalState ==
ConnectionState.Open) {
            MessageBox.Show("Là ca peut être très grave", "ERREUR",
MessageBoxButtons.OK, MessageBoxIcon.Error) ;
        }
    }
```

StateChange

Se produit chaque fois que la connexion passe d'ouverte à fermée ou inversement.

Sous ces airs anodins il est souvent intéressant d'utiliser cet événement. En effet trois cas peuvent être intéressants à gérer.

- Les erreurs de procédures stockées ferment la connexion.
- Cela permet de gérer un contrôle évènementiel de la fermeture effective de la connexion.
- Attendre l'ouverture réelle avant de manipuler les données.

Les arguments renvoyés sont l'état d'origine et le nouvel état.

Objet Transaction

Voilà une nouveauté ADO.NET qui est encore une amélioration par rapport à ADO.

Maintenant l'appel de BeginTrans crée un objet transaction qu'il est possible d'identifier. Cela permet de savoir quelles opérations sont incluses dans quelles transactions. Les propriétés de l'objet transaction sont une référence à la connexion qui supporte la transaction et le niveau d'isolation choisi.

Comme méthodes spécifiques, on trouve

Commit

Valide la totalité de la transaction.

RollBack

Annule la transaction jusqu'au dernier état stable. Celui-ci peut être soit la création de la connexion, soit le dernier point 'Save' que vous avez défini.

Save

Permet de fixer un point de la transaction limitant l'effet d'un RollBack.

L'exemple suivant montre une gestion assez complète d'une transaction.

```
SqlClient.SqlTransaction MaTrans ;
        cn.ConnectionString = @"packet size=4096;user id=sa;data
source=NOM-SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=northwind;password=monpasse";
        try {
            cn.Open();
            MaTrans = cn.BeginTransaction(IsolationLevel.Serializable,
"tr1");
            SqlClient.SqlCommand MaCommand = new SqlClient.SqlCommand();
            MaCommand.CommandText = "UPDATE Products SET UnitPrice =19
WHERE UnitPrice=20";
            MaCommand.Connection = cn ;
            MaCommand.Transaction = MaTrans ;
            MaCommand.ExecuteNonQuery();
            MaTrans.Save("tr1");
            MaCommand.CommandText = "UPDATE Products SET UnitPrice =19
WHERE UnitPrice=20";
        Catch(Exception ex) {
            MaTrans.Rollback();
        }
        MaTrans.Commit() ;
```

Les commandes

Elles sont identiques aux commandes ADO, avec maintenant la possibilité de ne pas utiliser de magasin de données pour récupérer le résultat d'une commande ne renvoyant qu'une valeur. Globalement il y a deux approches de l'objet commande.

- L'action sur la source de données attendant ou non des paramètres et renvoyant ou non des paramètres. Celle-ci regroupe aussi la possibilité de modifier la source à l'aide de commande SQL DDL.
- La recherche de données ou de résultats ne nécessitant pas la création d'un DataSet.

Les actions des commandes sont donc bidirectionnelles. Pour ma part j'ai toujours beaucoup apprécié ces objets dans ma programmation mais je sais que nombre de développeurs ne l'utilisent jamais afin d'écrire un code plus 'générique'.

Il existe des commandes gérées par le DataAdapter. Celles-ci sont stricto sensu des commandes ADO.NET. Nous en reparlerons dans la partie DataAdapter.

Propriétés

CommandText (String)

Définit le texte de la commande SQL, le nom de la table ou le nom de la procédure stockée.

CommandTimeOut (Integer)

Définit le temps avant la levée d'une exception si la commande n'a pas abouti.

CommandType

Constantes	Valeur	Description
Text	1	CommandText correspond à la définition textuelle d'une commande ou d'un appel de procédure stockée.
StoredProc	4	CommandText correspond au nom d'une procédure stockée.
TableDirect	512	CommandText correspond à un nom de table dont les colonnes sont toutes renvoyées.

La bonne définition de cette propriété est très importante.
Attention, TableDirect n'est supportée que par la commande OleDbCommand.

Connection(Connection)

Renvoie ou définit la connexion sur laquelle la commande est liée. Attention, vous lèverez une exception si la connexion est en cours d'extraction.

Transaction(Transaction)

Renvoie ou définit la transaction qui englobe la commande. La transaction définie doit forcément appartenir à la connexion définie dans la propriété Connection.

Parameters

Renvoie la collection des objets Parameter d'une commande. Voir plus loin l'utilisation dans 'Objet Parameter' du même chapitre.

UpdateRowSource

Spécifie comment la commande réagit lors d'une opération de modification (Update). Peut prendre une des valeurs suivantes :

Nom de membre	Description
Both	Les paramètres de sortie et la première ligne retournée sont mappés à la ligne modifiée dans DataSet.
FirstReturnedRecord	Les données de la première ligne retournée sont mappées à la ligne modifiée dans DataSet.
None	Tous les paramètres et les lignes retournés sont ignorés.
OutputParameters	Les paramètres de sortie sont mappés à la ligne modifiée dans DataSet.

Ceci est plutôt utilisé dans la mise à jours des données par les procédures stockées. C'est toutefois aussi applicable si vous travaillez avec des lots de requêtes.

Retour de procédure stockée

Discutons de l'utilisation de cette propriété. Lorsque j'écris une procédure stockée, je possède plusieurs possibilités d'action. Envisageons l'ajout d'une ligne dans la table 'Authors'. Celle-ci possède une colonne Auto-incrémentée pour le champ 'Au_Id' qui est la clé primaire, un champ texte 'Author' et un champ entier 'year born'. Une Procédure d'ajout va être du style

```
CREATE PROCEDURE InsertAuthor
        @Auteur  nchar(50),
        @Annee  integer
AS
        INSERT INTO Authors (Author, [year born])
                VALUES(@Auteur, @Annee)
RETURN @@ROWCOUNT
```

Cette procédure va fonctionner parfaitement, malheureusement elle est inutilisable en l'état. En effet, je ne passe pas de paramètres pour le champ 'Au_Id' puisque c'est le SGBD qui attribue la valeur. Seulement je ne renvoie pas la valeur créée ce qui fait qu'il sera impossible de mettre à jour la table.

Les administrateurs écrivent deux sortes de procédure pour gérer ce cas.

Utilisation d'un paramètre de sortie

```
CREATE PROCEDURE InsertAuthor
        @Auteur  nchar(50),
        @Annee  integer
       @Ident integer OUTPUT
AS
        INSERT INTO Authors (Author, [year born])
                VALUES(@Auteur, @Annee)
SET @ Ident = @@Identity
RETURN @@ROWCOUNT
```

Dans un cas comme celui-ci il suffit de mapper le paramètre de retour sur la ligne pour mettre la table à jour. Les paramètres de sortie sont simples à utiliser et plutôt rapides, mais @@IDENTITY pose des problèmes. Normalement @@IDENTITY renvoie la valeur générée par le SGBD à la dernière ligne ajoutée par votre connexion. Si jamais il existe un déclencheur (trigger) ajoutant une information dans une autre table utilisant un champ auto-incrémenté, c'est cette valeur que vous allez récupérer. Avec SQL-Server 2000 vous pouvez utiliser SCOPE_IDENTITY, mais sinon il y a un risque.

Utilisation d'une ligne de retour

Il existe aussi la possibilité de renvoyer la dernière ligne insérée par une requête SELECT. Dans ce cas, c'est la ligne renvoyée qu'il faut mapper. Notez bien que le problème de la récupération de la valeur générée reste identique. Imaginons le cas suivant :

```
CREATE PROCEDURE InsertAuthor
        @Auteur  nchar(50),
        @Annee  integer
       @Ident integer OUTPUT
AS
        INSERT INTO Authors (Author, [year born])
                VALUES(@Auteur, @Annee)
SET @ Ident = @@Identity
SELECT * FROM Authors WHERE Au_Id=@Ident
RETURN @@ROWCOUNT
```

Si un déclencheur a généré une autre valeur la ligne retournée peut être vide ou une ligne existante, mais pas celle que vous voulez. Pour récupérer une valeur fiable vous devez utiliser :

```
SELECT * FROM Authors WHERE Au_Id=@Ident AND Author=@Auteur
```

Dans ce cas, vous ne récupérerez pas de ligne, ce qui vous permettra de détecter l'erreur.

Nombres de lignes modifiées

Il s'agit là d'un problème un peu similaire, mais qui peut avoir des conséquences tout aussi graves. Dans le fonctionnement de la méthode Update du DataAdapter, il y a vérification du nombre de lignes modifiées pour savoir si la modification a échoué. Imaginons que l'objet InsertCommand du DataAdapter fait référence à la procédure stockée suivante :

```
CREATE PROCEDURE InsertAuthor
        @Auteur  nchar(50),
        @Annee  integer
       @Ident integer OUTPUT
AS
       INSERT INTO Authors (Author, [year born])
              VALUES(@Auteur, @Annee)
SET @ Ident = @@Identity
       INSERT INTO Audit(DateMod,Table,Enregistrement)
              VALUES(GETDATE(), 'Authors',@Ident)
RETURN @@ROWCOUNT
```

Il y aura toujours une insertion dans la table 'Audit'. La commande récupérera comme valeur soit 1 si l'insertion a échouée soit 2 si elle à réussie, mais en tout cas jamais zéro. Ceci fait qu'une insertion qui a échoué sera considérée comme ayant aboutie.

Pour ce genre de problème, il est indispensable d'utiliser la commande NOCOUNT correctement. Tout cela pour vous redire qu'il est indispensable de bien connaître les procédures stockées que l'on utilise.

Méthodes

Cancel

Fonctionne à l'identique de son homologue ADO, c'est à dire mal. Cancel tente d'annuler une commande en cours mais ne lève pas d'erreur si l'annulation échoue.

CreateParameter

Crée un nouvel objet Parameter. L'objet est créé mais n'est pas défini.

Je vais en profiter pour faire une digression sur les constructeurs. Avec les langages DotNet comme vous le savez sûrement, et sinon vous allez le savoir, on peut déclarer et affecter dans la même ligne. Les classes DotNet font toujours appel à un constructeur, mais celui-ci peut être surchargé, c'est à dire qu'il accepte plusieurs paramétrage différent. Prenons l'exemple de OleDbParameter. Il utilise le constructeur New (en C#) qui crée une nouvelle instance de la classe. Le constructeur accepte cinq surcharges :

public void New(String *name*, object *value*)
public void New(String *name*, OleDbType *dataType*)
public void New(String *name*, OleDbType *dataType*, int *size*)
public void New(String *name*, OleDbType *dataType*, int *size*, string *srcColumn*)
public void New(string *parameterName*, OleDbType *dbType*, int *size*, ParameterDirection *direction*, bool *isNullable*, byte *precision*, byte *scale*, string *srcColumn*, DataRowVersion *srcVersion*, object *value*)

De la même façon je peux dans la même ligne, instancier, définir et utiliser.

```
Command1.Parameters.Add(new OleDbParameter("Original_Au_ID",
OleDbType.Integer, 0, ParameterDirection.Input, false, (byte)0, (byte)0,
"Au_ID", DataRowVersion.Original, null));
```

Ce code est parfaitement juste mais peu lisible. Faites très attention avec ce type de codage.

ExecuteNonQuery

Execute la commande et renvoie le nombre de lignes affectées par la commande. C'est l'exécution classique d'une commande comme dans ADO. Si la commande est une requête SELECT, la valeur retournée est –1.

Il y a tout de même valorisation des paramètres de sortie et de retour.

ExecuteReader

Cette méthode sert à créer un objet DataReader. Cette méthode attend un paramètre CommandBehavior qui peut prendre les valeurs suivantes.

Nom	Valeur	Description
Default	0	équivaut fonctionnellement à l'appel à ExecuteReader().
SingleResult	1	La requête retourne un jeu de résultat unique.
SchemaOnly	2	La requête retourne uniquement les informations de colonne et n'affecte pas l'état de la base de données.
KeyInfo	4	La requête retourne des informations de colonne et de clé primaire. Elle est exécutée sans verrouiller les lignes sélectionnées.
SingleRow	8	La requête retourne normalement une ligne unique. L'exécution de la requête peut affecter l'état de la base de données. Certains fournisseurs de données .NET Framework peuvent éventuellement utiliser ces informations pour optimiser les performances de la commande. Lorsque vous spécifiez SingleRow avec la méthode **ExecuteReader** de l'objet **OleDbCommand**, le fournisseur de données.NET Framework pour OLE DB effectue la liaison à l'aide de l'interface IRow OLE DB si elle est disponible. Sinon, il utilise l'interface IRowset. Si votre instruction SQL doit normalement retourner une seule ligne, la spécification de SingleRow peut également améliorer les performances de l'application. Il est possible de spécifier SingleRow lorsque vous exécutez des requêtes qui retournent plusieurs jeux de résultats. Dans ce cas, plusieurs jeux de résultats sont toujours retournés, mais chaque jeu possède une ligne unique.
SequentialAccess	16	Fournit à DataReader un moyen de gérer les lignes qui contiennent des colonnes renfermant des valeurs binaires élevées. Plutôt que de charger la ligne entière, SequentialAccess permet à DataReader de charger les données en tant que flux. Vous pouvez ensuite utiliser la méthode GetBytes ou GetChars afin de spécifier un emplacement d'octets à partir duquel démarrer l'opération de lecture, ainsi qu'une taille limitée de mémoire tampon pour les données retournées. Lorsque vous spécifiez SequentialAccess, vous êtes obligé de lire les colonnes dans l'ordre où elles sont retournées, mais il n'est cependant pas nécessaire de lire chaque colonne. Lorsque vous avez lu au-delà d'un emplacement du flux de données retourné, les données situées jusqu'à cet emplacement ne peuvent plus être lues à partir de DataReader. Si vous utilisez OleDbDataReader, vous pouvez relire la valeur de la colonne en cours jusqu'à ce que vous lisiez au-delà de celle-ci. Si vous utilisez SqlDataReader, une valeur de colonne peut être lue à une seule reprise.
CloseConnection	32	Lorsque la commande est exécutée, l'objet Connection associé se ferme en même temps que l'objet DataReader.

L'appel de l'objet DataReader renvoie normalement un curseur de type FireHose. Rappelons ici ses principales caractéristiques. Le DataReader va être en lecture seule. Il ne peut se déplacer que vers l'avant, d'une ligne à la fois. Seul l'enregistrement lu se trouve dans votre cache. Ceci fait que les modifications engendrées par d'autres utilisateurs sur des enregistrements non encore lus seront prises en compte. Il n'est pas possible avec un tel objet de déterminer combien de lignes seront effectivement renvoyées.

Bien que nous voyions plus en détail son utilisation plus loin, voilà la gestion classique d'un DataReader.

```
this.Button1.Click += new System.EventHandler(Button1_Click);
…
private void Button1_Click(object sender, System.EventArgs e) {
        SqlClient.SqlDataReader dtaRead;
        SqlClient.SqlCommand
MaCommand = new SqlClient.SqlCommand();
        cn.ConnectionString = @"packet size=4096;user id=sa;data
source=NOM-SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=northwind;password=monpasse";
        cn.Open();
        MaCommand.CommandText = "SELECT * FROM Products";
        MaCommand.CommandType = CommandType.Text;
        MaCommand.Connection = cn;
        dtaRead =
MaCommand.ExecuteReader(CommandBehavior.CloseConnection);
        while(dtaRead.Read()) {
                this.ComboBox1.Items.Add(dtaRead.GetString(1));
        }
}
```

Notez qu'il faut toujours faire au moins un appel à Read car le positionnement du DataReader à sa création est avant le premier enregistrement.

ExecuteScalar

Cette méthode permet de récupérer un résultat unique sans créer de jeu de données. Elle est spécialement utilisée pour récupérer le résultat d'une requête d'agrégation.

```
this.Button1.Click += new System.EventHandler(Button1_Click);
…
private void Button1_Click(object sender, System.EventArgs e) {
        SqlClient.SqlCommand MaCommand = new SqlClient.SqlCommand();
        cn.ConnectionString = @"packet size=4096;user id=sa;data
source=NOM-SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=northwind;password=monpasse";
        cn.Open();
        MaCommand.CommandText = "SELECT Count(*) FROM Products";
        MaCommand.CommandType = CommandType.Text;
        MaCommand.Connection = cn;
        MessageBox.Show(.ExecuteScalar());
}
```

Il y a un intérêt certain à ne pas créer de jeux d'enregistrements pour obtenir une seule réponse. Cependant il convient de ne pas remplacer la création d'un jeu de données, DataReader ou Dataset par la multiplication d'appel ExecuteScalar.

Prepare

Identique dans ADO. Ceci consiste à compiler une requête avant de l'exécuter. Certaines règles doivent s'appliquer avant de décider d'une telle action :

- Une requête exécutée une fois ne doit pas être compilée
- L'ensemble des paramètres doit être correctement défini avant de préparer une commande. Une définition correcte sous-entend que chaque paramètre possède un type et que chaque paramètre ayant un type de longueur variable ait une taille définie.
- La connexion sur laquelle s'appuie la commande doit être définie et ouverte.

Objet Parameter

Dans cette partie et dans la suivante, nous allons regarder la manipulation des paramètres dans ADO.NET. Il convient de concevoir ce qu'on entend par paramètres. Il existe trois types de paramètres dont deux sont uniquement utilisés dans le cadre des procédures stockées.

Les paramètres d'entrée : Ce sont ceux que vous communiquez à votre requête / procédure afin qu'elle puisse fonctionner.

Les paramètres de sortie : Ils sont définis par une procédure stockée. La récupération de leurs valeurs n'a de sens qu'après exécution mais il est parfois nécessaire d'interroger la procédure pour connaître leurs types ou leurs tailles.

Les paramètres de retour : Il s'agit d'une valeur parfois renvoyer par l'instruction RETURN. En général, on utilise ce paramètre pour indiquer l'état de succès de la procédure.

Les objets paramètres appartiennent à un espace de nom, nous trouverons donc selon les cas des objets SqlParameter, OleDBParameter, etc…

Les paramètres appartiennent à une collection qui est liée soit à la commande, soit au mappage d'un Dataset dans le DataAdapter. Une astuce classique consiste à se rappeler que les méthodes appartiennent à la collection et les propriétés au paramètre.

Il y a deux possibilités d'utilisation selon les fournisseurs.

Utiliser des paramètres nommés

C'est le mode obligatoire pour travailler avec le fournisseur SQL-Server. Attention cela ne veut pas dire que vous ne pouvez pas passer par le mode ordinal dans ADO.NET. Cela veut juste dire que le texte de la commande doit utiliser des paramètres nommés et non des espaces réservés (?). Les paramètres peuvent être ajoutés à la collection dans l'ordre de votre choix.

Utiliser des espaces réservés

C'est le mode privilégié du fournisseur OLEDB. Attention, certains SGBD n'acceptent pas cette syntaxe et forcent à l'utilisation de paramètres nommés. Les espaces réservés attendent une définition dans l'ordre de leur apparition. Il n'existe aucune dérogation à cette règle puisque les paramètres n'ont pas de noms.

```
OleDbCommand MaCommand = new OleDbCommand("UPDATE Authors SET Author=?,
[year born]=? WHERE Au_Id = ?", MaConn) ;
MaCommand.Parameters.Add("Au_Id", OleDbType.Integer).Value = 1;
MaCommand.Parameters.Add("Annee", OleDbType.SmallInt).Value = 1967;
MaCommand.Parameters.Add("Auteur", OleDbType.VarChar, 50).Value =
"RABILLOUD";
int Retour = MaCommand.ExecuteNonQuery();
MessageBox.Show(Retour.ToString);
```

Dans ce cas la valeur de retour va être 0 mais aucune exception ne sera levée.

```
OleDbCommand MaCommand = new OleDbCommand("UPDATE Authors SET Author=?,
[year born]=? WHERE Au_Id = ?", MaConn) ;
MaCommand.Parameters.Add("Auteur", OleDbType.VarChar, 50).Value =
"RABILLOUD";
MaCommand.Parameters.Add("Annee", OleDbType.SmallInt).Value = 1967;
MaCommand.Parameters.Add("Au_Id", OleDbType.Integer).Value = 1;
int Retour = MaCommand.ExecuteNonQuery();
MessageBox.Show(Retour.ToString);
```

Par contre, cette commande va fonctionner correctement puisque les paramètres sont bien ajoutés dans le même ordre que leur apparition dans la commande.

Méthodes de ParameterCollection

Add

Il existe plusieurs versions surchargées de cette méthode. Dans le principe on ajoute toujours un objet paramètre à la collection. On peut faire la définition dans la même ligne que l'ajout mais ce n'est pas recommandable.

public override SqlParameter Add(String parameterName**, SqlDbType** sqlDbType**, int** size**, String** sourceColumn**)**

Clear

Supprime les paramètres de la collection.

Contains

Vérifie la présence d'un paramètre dans une commande. Existe dans deux syntaxes permettant de chercher le paramètre par son nom ou par sa valeur.

CopyTo

Fait une copie des paramètres d'une collection dans un tableau.

IndexOf

Renvoie la position ordinale d'un paramètre dans la collection. Accepte le nom ou l'objet paramètre comme argument.

Insert

Insère un objet paramètre à la position indiquée

Remove

Enlève le paramètre passé en argument.

RemoveAt

Similaire à la précédente méthode, mais accepte comme argument la position ordinale ou le nom du paramètre.

Propriétés des paramètres

DBType (Dbtype)

Définit le type du paramètre. Il convient de faire attention car si celui-ci n'est pas défini, il y aura définition d'un type en partant de la valeur du paramètre. Ce comportement présente un risque élevé quant au fonctionnement de l'application. Le tableau de mappage suivant résume les types à utiliser.

Type .NET Framework	DbType	SqlDbType	OleDbType	OdbcType	OracleType
bool	Boolean	Bit	Boolean	Bit	Byte
byte	Byte	TinyInt	UnsignedTinyInt	TinyInt	Byte
byte[]	Binary	VarBinary. Cette conversion implicite échouera si le tableau d'octets est supérieur à la taille maximale de VarBinary, soit 8 000 octets.	VarBinary	Binary	Raw
char		La déduction de SqlDbType à partir de char n'est pas prise en charge.	Char	Char	Byte
DateTime	DateTime	DateTime	DBTimeStamp	DateTime	DateTime
Decimal	Decimal	Decimal	Decimal	Numeric	Number
double	Double	Float	Double	Double	Double
float	Single	Real	Single	Real	Float
Guid	Guid	UniqueIdentifier	Guid	UniqueIdentifier	Raw
Int16	Int16	SmallInt	SmallInt	SmallInt	Int16
Int32	Int32	Int	Int	Int	Int32
Int64	Int64	BitInt	BigInt	BigInt	Number
object	Object	Variant	Variant	La déduction de OdbcType à partir de Object n'est pas prise en charge.	Blob
string	String	NVarChar. Cette conversion implicite échouera si la chaîne est supérieure à la taille maximale de NVarChar, soit 4 000 caractères.	VarWChar	NVarChar	NVarChar
Timespan	Time	La déduction de SqlDbType à partir de TimeSpan n'est pas prise en charge.	DBTime	Time	DateTime
UInt16	UInt16	La déduction de SqlDbType à partir de UInt16 n'est pas prise en charge.	UnsignedSmallInt	Int	UInt16
UInt32	UInt32	La déduction de SqlDbType à partir de UInt32 n'est pas prise en charge.	UnsignedInt	BigInt	UInt32

Type .NET Framework	DbType	SqlDbType	OleDbType	OdbcType	OracleType
UInt64	UInt64	La déduction de SqlDbType à partir de UInt64 n'est pas prise en charge.	UnsignedBigInt	Numeric	Number
	AnsiString	VarChar	VarChar	VarChar	VarChar
	AnsiStringFixedLength	Char	Char	Char	Char
	Currency	Money	Currency	La déduction de OdbcType à partir de Currency n'est pas prise en charge.	Number
	Date	La déduction de SqlType à partir de Date n'est pas prise en charge.	DBDate	Date	DateTime
	SByte	La déduction de SqlType à partir de SByte n'est pas prise en charge.	TinyInt	La déduction de OdbcType à partir de SByte n'est pas prise en charge.	SByte
	StringFixedLength	NChar	WChar	NChar	NChar
	Time	La déduction de SqlType à partir de Time n'est pas prise en charge.	DBTime	Time	DateTime
	VarNumeric	La déduction de SqlDbType à partir de VarNumeric n'est pas prise en charge.	VarNumeric	La déduction de OdbcType à partir de VarNumeric n'est pas prise en charge.	Number

Direction (ParameterDirection)

Peut prendre une des valeurs suivantes :

Nom de membre	Description
Input	Le paramètre est un paramètre d'entrée.
InputOutput	Le paramètre est à la fois un paramètre d'entrée et de sortie.
Output	Le paramètre est un paramètre de sortie.
ReturnValue	Le paramètre représente une valeur de retour d'une opération telle qu'une procédure stockée, une fonction intégrée ou une fonction définie par l'utilisateur.

IsNullable (Booléen)

Indique si le paramètre peut prendre la valeur NULL.

Offset (SQL parameter uniquement)

Utilisée avec les types Long Binary et Long String. Indique la taille des fragments passés au client.

Precision (Byte)

Uniquement si le type est 'Decimal'. Définit le nombre de chiffres utilisés pour donner la valeur. Elle ne peut être séparée de la définition de la propriété 'Scale'. La version 1.0 du FrameWork ne vérifie pas les valeurs de ces deux propriétés.

Scale (Byte)

Définit le nombre de décimal d'une donnée de type décimal. Liée à la propriété précision.

Size (Integer)

Définie la taille en octet d'un paramètre d'entrée. N'est utilisée qu'avec des types de données de longueur variable (chaîne ou binaire). Là encore la taille sera déduite si elle n'est pas définie.

SourceColumn (String)

Utilise les informations de mappage pour attribuer une valeur au paramètre.

C'est en fait comme cela que travaillent généralement les commandes de mise à jour du DataAdapter.

Regardons le code suivant :

```
//OleDbUpdateCommand
        Command1.CommandText = "UPDATE Authors SET Author = ?, [Year
Born] = ? WHERE (Au_ID = ?) AND (Author = ? " & _
        "OR ? IS NULL AND Author IS NULL) AND ([Year Born] = ? OR ? IS
NULL AND [Year Bor" & _
        "n] IS NULL)";
        Command1.Connection = Me.OleDbConnection1;
        Command1.Parameters.Add(new OleDbParameter("Author",
OleDbType.VarWChar, 50, "Author"));
        Command1.Parameters.Add(new OleDbParameter("Year_Born",
OleDbType.SmallInt, 0, "Year Born"));
        Command1.Parameters.Add(new OleDbParameter("Original_Au_ID",
OleDbType.Integer, 0, ParameterDirection.Input, false, (byte)0,
(byte)0,""Au_ID", DataRowVersion.Original, null));
        Command1.Parameters.Add(new OleDbParameter("Original_Author",
OleDbType.VarWChar, 50, ParameterDirection.Input, false, (byte)0,
(byte)0," "Author", DataRowVersion.Original, null));
        Command1.Parameters.Add(new OleDbParameter("Original_Author1",
OleDbType.VarWChar, 50, ParameterDirection.Input, false, (byte)0,
(byte)0," "Author", DataRowVersion.Original, null));
        Command1.Parameters.Add(new OleDbParameter("Original_Year_Born",
OleDb.OleDbType.SmallInt, 0, ParameterDirection.Input, false, (byte)0,
(byte)0,""Year Born", DataRowVersion.Original, null));
        Command1.Parameters.Add(new OleDbParameter("Original_Year_Born1",
OleDbType.SmallInt, 0, ParameterDirection.Input, false, (byte)0,
(byte)0,"Year Born", DataRowVersion.Original, null));
```

Le texte de la commande est de la forme :

```
Command1.CommandText = "UPDATE Authors SET Author = ?, [Year Born] = ?
WHERE (Au_ID = ?) AND (Author = ? OR ? IS NULL AND Author IS NULL) AND
([Year Born] = ? OR ? IS NULL AND [Year Born] IS NULL)";
```

Je dois donc passer sept paramètres à ma commande. Il serait fastidieux de gérer à la volée tous ces paramètres, nous allons donc utiliser les propriétés des paramètres.

Pour un paramètre passant une des valeurs nouvelles, la syntaxe est :

```
Command1.Parameters.Add(New OleDbParameter("Author", OleDbType.VarWChar,
50, "Author"));
```

Soit de la forme :

OleDbParameter Add(String *parameterName***, OleDbType** *oleDbType***, int** *size***,String** *sourceColumn***)**

Pour le paramètre devant passer l'ancienne valeur du même champ

```
Command1.Parameters.Add(new OleDbParameter("Original_Author",
OleDbType.VarWChar, 50, ParameterDirection.Input, false, (byte)0,
(byte)0,"Author", DataRowVersion.Original, null));
```

Soit de la forme :

New(String *parameterName***, OleDbType** *dbType***, Integer** *size***, ParameterDirection** *direction***, bool** *isNullable***, byte** *precision***, byte** *scale***, String** *srcColumn***, DataRowVersion** *srcVersion***, object** *value***)**

SourceVersion (DataRowVersion)

Définit la version de la colonne désignée par SourceColumn pour valoriser le paramètre. Peut prendre une des valeurs suivantes :

membre	Description
Current	Le paramètre utilise la valeur actuelle de la colonne. Il s'agit de la valeur par défaut.
Default	Le paramètre utilise le DefaultValue de la colonne.
Original	Le paramètre utilise la valeur d'origine de la colonne.
Proposed	Le paramètre utilise une valeur proposée.

La classe DataReader

Nous avons donc vu que l'objet DataReader est obtenu par l'intermédiaire d'un objet Command et d'un objet Connection. Dans le cas de cette classe, vous ne pouvez pas utiliser directement un constructeur, il est obligatoire d'appeler la méthode ExecuteReader.

L'objet DataReader est exclusif sur la connexion, tout comme l'est le curseur équivalent dans ADO. Ceci revient à dire que la connexion utilisée par le service du DataReader ne supportera aucun autre composant. Seul peut être appelé sa méthode Close. La connexion est libérée si vous fermez l'objet DataReader.

Il y a levée d'une exception sur l'appel d'une connexion déjà utilisée par un DataReader.

La visibilité des modifications induites par d'autre processus est assez difficile à percevoir. Disons que les enregistrements supprimés ou modifiés n'ayant pas encore été lus seront vus avec les modifications. Il faut donc se méfier de la possibilité d'avoir une lecture partiellement incohérente. J'entends par-là qu'il est tout à fait possible qu'une opération ait lieu pendant la lecture et affectant des enregistrements déjà lus et d'autres non.

Le DataReader est idéal pour le parcours unique d'un jeu d'enregistrement, à des fins de remplissage par exemple. Il est soumis aux règles des objets connectés, ce qui veut dire qu'il doit être utilisé le moins longtemps possible. N'oubliez jamais que c'est le serveur qui fournit le curseur, n'abusez donc pas de ces objets sur un même serveur.

Le code suivant est un exemple standard de manipulation du DataReader.

```
this.Form1.Load += new System.EventHandler(Form1_Load);
    …
    private void Form1_Load(object sender, System.EventArgs e) {
        SqlClient.SqlConnection MaConn = new
SqlClient.SqlConnection(@"packet size=4096;user id=sa;data source=NOM-
SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=pubs;password=monpasse");
        MaConn.Open();
        string strSQL = "SELECT Author FROM Authors";
        SqlClient.SqlCommand MaComm = new SqlClient.SqlCommand(strSQL,
MaConn);
        MaComm.CommandType = CommandType.Text;
        SqlClient.SqlDataReader dtaRead = MaComm.ExecuteReader();
        while(dtaRead.Read())
            this.ComboBox1.Items.Add(dtaRead.GetSqlString(0));
        }
        dtaRead.Close();
    }
```

Propriétés

FieldCount (Integer)

Renvoie le nombre de colonnes (champ) présent dans la ligne en cours.

HasRows (Booléen)

Renvoie vrai si le jeu d'enregistrements contient au moins une ligne.

IsClosed (Booléen)

Renvoie vrai si l'objet est fermé.

Item

Renvoie la valeur de la colonne dans son format natif. On peut identifier la colonne par son nom ou par sa position ordinale. Il est toujours meilleur pour les performances de préférer la position ordinale. Item peut toujours être omis, mais la syntaxe ainsi obtenue est particulièrement absconse.

RecordsAffected (Integer)

Cette propriété est particulière à plus d'un point. En effet il n'est pas simple de comprendre ce qui peut affecter des enregistrements avec un curseur en lecture seule. Cette propriété renvoie le nombre d'enregistrements affectés avec l'emploi d'une procédure stockée ou d'un traitement SQL par lot comprenant des requêtes actions. Dans la plupart des cas elle renverra –1. Il faut accéder à cette valeur après fermeture de DataReader autant que faire se peut pour obtenir un résultat fiable.

Méthodes

Dans la plupart des méthodes de récupération de valeur, il faut désigner la colonne cible par son numéro ordinal.

Close

Ferme le DataReader. Ceci libère la connexion. Seule les propriétés IsClosed et RecordsAffected sont encore accessibles après la fermeture.

Attention, dans le cas d'une procédure stockée, les paramètres de sortie ne sont valorisés qu'après l'appel de la méthode Close.

GetBoolean, GetByte, GetChar, etc….

Renvoie la valeur d'une colonne sous la forme d'une donnée typée telle que demandée dans la méthode Get typée.

Il n'y a pas conversion effective de la valeur, vous devez donc être sûr que la donnée sera bien du type spécifié. On utilise souvent la méthode IsDbNull avant.

GetDataTypeName, GetFieldType

Renvoie le type ou le nom du type de la colonne désignée.

GetName, GetOrdinal

Renvoie le nom de la colonne en fonction de sa position ou inversement.

GetSchemaTable

Permet de récupérer les informations de colonnes (méta-données) dans un objet DataTable. Cette méthode implique donc la création d'un DataSet. C'est partiellement l'équivalent de la méthode OpenSchema ADO. Vous devez passer KeyInfo comme paramètre de la méthode ExecuteReader pour que les informations renvoyées soient correctes.

Les informations renvoyées sont :

Colonne DataReader	Description
ColumnName	Nom de la colonne. S'il ne peut pas être déterminé, une valeur Null est retournée. Ce nom reflète toujours le dernier nom attribué à la colonne (Alias par exemple).
ColumnOrdinal	Numéro de la colonne. Il s'agit de zéro pour le signet de colonne de la ligne, le cas échéant. Les autres colonnes sont numérotées à partir de un. Cette colonne ne peut pas contenir de valeur Null.
ColumnSize	Longueur maximale possible pour une valeur de la colonne. Pour les colonnes qui utilisent un type de données de longueur fixe, il s'agit de la taille du type de données.
NumericPrecision	Si ProviderType est un type de données numérique, il s'agit de la précision maximale de la colonne. La précision dépend de la définition de la colonne. Sinon, il s'agit d'une valeur Null.
NumericScale	Si ProviderType est DBTYPE_DECIMAL ou DBTYPE_NUMERIC, il s'agit du nombre de chiffres à droite de la virgule décimale. Sinon, il s'agit d'une valeur Null.
IsUnique	true : deux lignes figurant dans la table retournée dans BaseTableName ne peuvent pas avoir la même valeur dans cette colonne ; c'est à dire si la colonne constitue une clé en soi ou s'il existe une contrainte de type UNIQUE qui s'applique uniquement à cette colonne. false : La colonne peut contenir des valeurs dupliquées dans la table de base. La valeur par défaut de cette colonne est false.
IsKey	true : la colonne fait partie d'un ensemble de colonnes du jeu de lignes qui, dans son ensemble, identifie de manière unique la ligne. L'ensemble de colonnes avec IsKey égal à true doit identifier de manière unique une ligne du jeu de lignes. Ce jeu de colonnes ne doit pas obligatoirement être un jeu minimal de colonnes. Ce jeu de colonnes peut être généré à partir de la clé primaire d'une table de base, d'une contrainte unique ou d'un index unique. false : la colonne n'est pas tenue d'identifier de manière unique la ligne.
BaseServerName	Le nom de l'instance de Microsoft SQL Server utilisée par SqlDataReader.
BaseCatalogName	Nom du catalogue dans le magasin de données qui contient la colonne. NULL si le nom du catalogue de base ne parvient pas à être déterminé.
BaseColumnName	Nom de la colonne dans le magasin de données. Il peut être différent du nom de colonne retourné dans la colonne ColumnName si un alias a été utilisé. Valeur Null si le nom de la colonne de base ne peut pas être déterminé ou si la colonne du jeu de lignes est dérivée d'une colonne du magasin de données, et non identique à celle-ci.
BaseSchemaName	Nom du schéma dans le magasin de données qui contient la colonne. Valeur Null si le nom du schéma de base ne peut pas être déterminé.
BaseTableName	Nom de la table ou de la vue dans le magasin de données qui contient la colonne. Valeur Null si le nom de la table de base ne peut pas être déterminé.
DataType	Type de la colonne au type .NET Framework
AllowDBNull	Défini si le consommateur ne peut pas affecter de valeur Null à la colonne ou si le fournisseur ne parvient pas à déterminer si le consommateur peut affecter une valeur Null à la colonne. Non défini dans les autres cas. Une colonne peut contenir des valeurs Null, même si une valeur Null ne peut pas lui être affectée.
ProviderType	Indicateur du type de données de la colonne. Si le type de données de la colonne varie d'une ligne à l'autre, il doit s'agir de la valeur Object. Cette colonne ne peut pas contenir de valeur Null.

Colonne DataReader	Description
IsAliased	true si le nom de la colonne est un alias, sinon, false.
IsExpression	true si la colonne est une expression, sinon, false.
IsIdentity	true si la colonne est une colonne identité, sinon, false.
IsAutoIncrement	true : la colonne assigne des valeurs aux nouvelles lignes selon des incréments fixes. false :La colonne n'assigne pas des valeurs aux nouvelles lignes par incréments fixes. La valeur par défaut de cette colonne est false.
IsRowVersion	Définit si la colonne contient un identificateur de ligne persistant dans lequel il est impossible d'écrire et ne possède aucune valeur significative, excepté pour identifier la ligne.
IsHidden	true si la colonne est masquée, sinon, false.
IsLong	Défini si la colonne comporte un objet binaire volumineux (BLOB, Binary Long OBject) comportant des données très volumineuses. La définition des données très volumineuses est propre au fournisseur.
IsReadOnly	true si la colonne ne peut pas être modifiée, sinon, false.

GetSqlBinary, GetSqlBoolean, etc…

Permet de récupérer les données sous forme de types mappés SqlType. Les données décrites dans ces types sont plus facilement manipulable (voir avant Type Sql).

GetValue, GetSqlValue

Récupère un objet représentant la donnée soit dans le type Framework, soit dans le type SQL.

GetValues, GetSqlValues

Renvoient un tableau d'objets contenant toutes les données de la ligne.

IsDBNull

Renvoie vrai si la donnée est Null. En fait s'il s'agit d'un objet DBNull. Cet objet permet de travailler comme dans ADO puisqu'il sait faire la différence entre une valeur Null (VT_NULL) et une ligne non initialisée (VT_EMPTY). Les opérations avec Null suivent les règles classiques.

NextResult

Permet de changer le jeu de résultat actif dans les opérations par lot.
Imaginons le cas suivant. Je vais appeler la procédure suivante :

```
CREATE PROCEDURE EssaiParamSortie (@ParSortie Integer OUTPUT) AS
SELECT @ParSortie = Count(Au_Id) FROM Authors;
SELECT au_lname FROM Authors;
GO
```

Cette procédure va nous permettre de comprendre la valorisation des paramètres de sortie par un DataReader. Notez qu'il a été écrit pour le fournisseur managé pour SQL-Server.

```
this.Form1.Load += new System.EventHandler(Form1_Load);
    …
    private void Form1_Load(object sender, System.EventArgs e) {
        SqlClient.SqlConnection MaConn = new
SqlClient.SqlConnection(@"packet size=4096;user id=sa;data source=NOM-
SZ71L7KTH17\\TUTO;persist security info=False;initial
catalog=pubs;password=monpasse");
        MaConn.Open();
        string strSQL = "EssaiParamSortie";
        SqlClient.SqlCommand MaComm = new SqlClient.SqlCommand(strSQL,
MaConn);
        MaComm.CommandType = CommandType.StoredProcedure;
        SqlClient.SqlParameter MonParam = new SqlClient.SqlParameter();
        MonParam.ParameterName = "@ParSortie";
        MonParam.Direction = ParameterDirection.Output;
        MonParam.DbType = DbType.Int32;
        MaComm.Parameters.Add(MonParam);
        SqlClient.SqlDataReader dtaRead = MaComm.ExecuteReader();
        MessageBox.Show((string)(MonParam.Value));
        do{
            while(dtaRead.Read())
                this.ComboBox1.Items.Add(dtaRead.GetSqlString(0));
            }
            MessageBox.Show((string)(MonParam.Value));
        }while(dtaRead.NextResult());
        MessageBox.Show((string)(MonParam.Value));
        dtaRead.Close();
        MessageBox.Show((string)(MonParam.Value));
    }
```

Vous verrez que la valeur du paramètre n'est correctement documentée qu'à la sortie de la boucle NextResult.

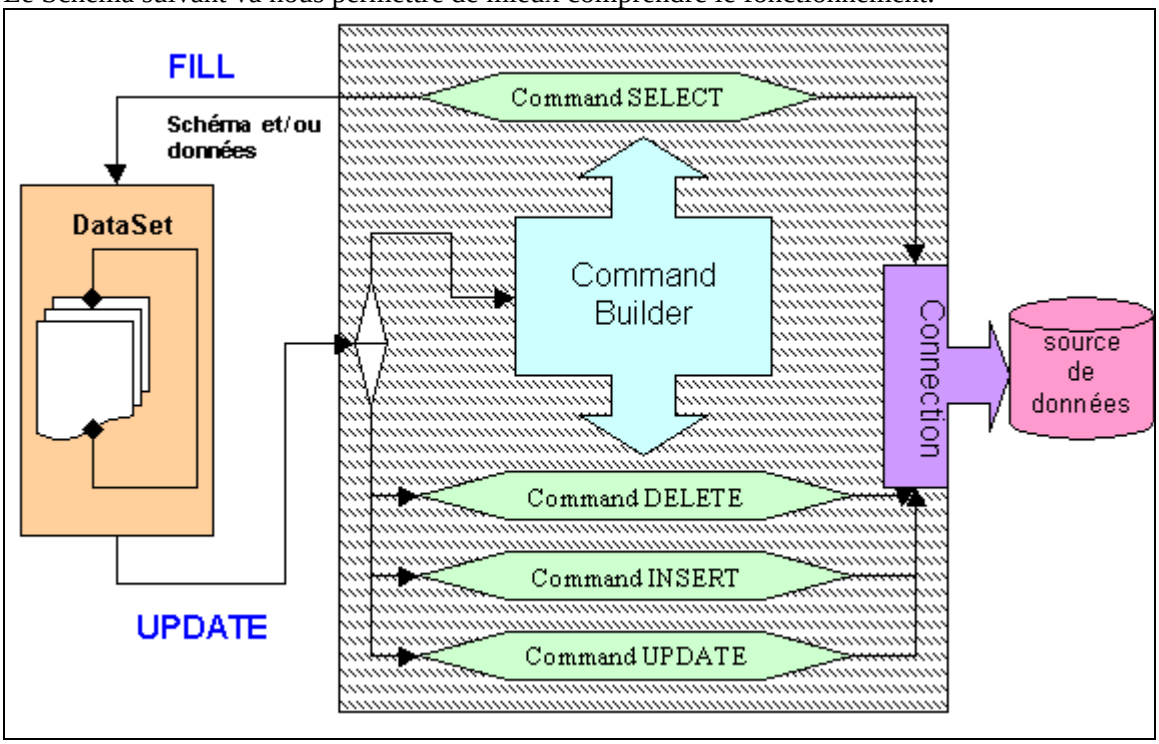
Read

Cette méthode avance le curseur jusqu'à l'enregistrement suivant. Elle renvoie True s'il reste des enregistrements après, sinon False. Attention l'appel de Read ne signifie pas la récupération des valeurs.

La classe DataAdapter

Pour appréhender correctement le DataAdapter, il faut le voir dans une certaine mesure comme un moteur de curseur configurable. En effet, il échange des informations entre la source et le jeu de données. On peut l'utiliser de façon assez transparente ou configurer finement son paramétrage afin de savoir exactement quelles données et méta-données seront renvoyées.

Le Schéma suivant va nous permettre de mieux comprendre le fonctionnement.



Comme nous l'avons déjà dit, il n'y a jamais de connexion directe entre le DataSet et la source de données. Le DataSet ne peut communiquer qu'avec le DataAdapter à l'aide des méthodes que celui-ci fournit. Le DataAdapter quant à lui communique avec la source en utilisant un objet Connection et quatre objets Command. On peut dire qu'à minima, il faut qu'il y ait une commande Select et une connexion à un DataAdapter pour fonctionner.

Plusieurs types de requêtes sont comprises dans les méthodes Fill selon que l'on souhaite récupérer des données, des méta-données ou les deux. De la même manière, l'appel de la méthode Update peut entrainer des actions diverses selon le paramétrage. Néanmoins une chose est invariable, il y a toujours utilisation des objets Command pour transmettre les actions.

L'objet CommandBuilder que vous voyez permet de laisser le DataAdapter créer les objets Command Action en fonction des informations qu'il trouve dans l'objet Command Select. Les objets Command utilisés par le DataAdapter peuvent être des requêtes SQL, des noms de tables, des procédures stockées…

Habituellement, un DataAdapter ne prends en charge qu'une table d'un DataSet, mais cela n'est pas une obligation.

Schéma et Mappage

Il faut ici déterminer ici ces termes pour que nous nous comprenions correctement.

- Le mappage est la lecture du schéma **de la source de données** que fait le DataAdapter.
- Le schéma est la lecture du schéma **du DataSet** que fait le DataAdapter.

Un DataAdapter contient une collection TableMappings qui lui permet de faire le lien entre le schéma de la source et le schéma du DataSet. Généralement les deux sont identiques, mais il arrive que non. Lorsqu'il n'y a pas de liaison entre les deux, le DataAdapter peut rencontrer des problèmes lors des tentatives de mises à jour. Pour éviter des erreurs malheureusement fréquentes vous devez soit gérer entièrement le mappage par le code, soit laisser le DataAdapter le gérer entièrement.

Propriétés

AcceptChangesDuringFill (Booléen)

Valeur par défaut à True. Indique si l'état des enregistrements (DataRow) passe à 'Unchanged' lors d'une opération Fill ou si celui-ci reste dans son état de modification.

Cette propriété ne doit pas être modifiée lors du premier Fill sauf dans le cas où vous souhaitez récupérer une table d'une source de donnée pour l'insérer dans une autre source. Lorsque vous procéder à plusieurs opérations Fill successives, il peut être intéressant de passer cette propriété à False afin de détecter les enregistrements ayant subi des modifications du fait d'autres utilisateurs.

ContinueUpdateOnError (Booléen)

Stipule le comportement lors d'une erreur de mise à jour. Valeur par défaut à False.

Nous reviendrons sur cette propriété plus loin, car elle définit un choix stratégique dans la mise à jour d'une source de données. Si sa valeur est vraie, il n'y a pas de levée d'exception lors d'une erreur de mise à jour d'une ligne. La propriété RowError récupère les valeurs d'erreurs et le traitement continu.

MissingMappingAction

Détermine le comportement du DataAdapter au cas ou des informations de mappage manquent. Voir plus loin TableMapping. Peut prendre une des valeurs suivantes :

Nom	Description
Error	Une exception InvalidOperationException est générée si le mappage de la colonne spécifiée est manquant.
Ignore	Toute colonne ou table ne possédant pas de mappage est ignorée. Retourne une référence Nothing dans Visual Basic.
Passthrough	La table ou colonne source est créée et ajoutée à DataSet à l'aide de son nom d'origine.

MissingSchemaAction

Détermine le comportement du DataAdapter au cas où les informations de schéma manquent. Peut prendre une des valeurs suivantes :

Nom	Description
Add	Ajoute les colonnes nécessaires pour achever le schéma.
AddWithKey	Ajoute les colonnes et les informations de clé primaire nécessaires pour achever le schéma. L'utilisateur peut également définir de manière explicite les contraintes de clé primaire sur chaque DataTable. De cette façon, les enregistrements entrants qui correspondent à des enregistrements existants sont mis à jour plutôt que d'être ajoutés.
Error	Une exception InvalidOperationException est générée si le mappage de colonnes spécifiées est manquant.
Ignore	Ignore les colonnes supplémentaires.

TableMappings

C'est la collection qui représente le mappage effectué par le DataAdapter. Rappelez-vous que le mappage concerne une source de données et un objet DataSet. Si vous avez laissé le DataAdapter gérer le mappage, il existe un objet nommé 'Table' dans cette collection. Sinon, c'est à vous de la remplir correctement. Par exemple :

```
DataTableMapping PersMap = dtaAdapt.TableMappings.Add("Authors",
"Auteurs");
dtaAdapt.ColumnMappings.Add( "Au_Id", "NumAut");
dtaAdapt.ColumnMappings.Add( "Author", "Auteur");
dtaAdapt.ColumnMappings.Add( "year born", "Annee");
```

Le mappage doit être fait avant l'appel d'une méthode Fill ou Update sinon le DataAdapter fera son mappage par défaut. Ne vous inquiétez pas si vous trouvez tout cela un peu abscons, nous verrons un exemple détaillé dans la troisième partie.

Méthodes

Fill

C'est donc la méthode de récupération des données vers un DataSet. Il existe plusieurs versions surchargées, mais seules deux sont généralement utilisées selon que la cible sera un DataTable simple ou un DataSet plus complexe.

La syntaxe est de la forme :

public override int Fill(DataSet **NomDataSet,** String ***NomDataTable*****);**

La valeur retournée est le nombre de lignes rapatriées. Si DataTable existe déjà les lignes existantes sont mises à jour.

Cette méthode fait appel à l'objet Command SELECT du DataAdapter. Celui-ci doit contenir une commande valide. Si la connexion utilisée est fermée, la méthode Fill l'ouvre le temps du rapatriement et la referme, sinon la connexion reste ouverte après le remplissage.

Si la commande renvoie plusieurs jeux de résultats, il sera créé autant de table que de jeux. Si une erreur survient pendant l'opération, les données déjà rapatriées seront gardées et l'opération sera arrêtée.

Lorsqu'on utilise la méthode Fill pour actualiser des données, la commande SELECT ne doit pas avoir été modifiée. De plus, il faut que la ou les colonnes définissant la clé primaire doivent être renseignées. Si tel n'est pas le cas, les lignes seront dupliquées. Si la requête est une requête avec jointure vous devrez créer vous-mêmes la clé.

FillSchema

Méthode de récupération des informations de schéma. Lors de l'utilisation de la méthode Fill, seuls les noms de colonnes sont récupérés. Si vous voulez récupérer des informations de schéma, vous devez utiliser cette méthode

public override DataTable[] FillSchema(DataSet ***NomDataSet,*** SchemaType ***TypeSchema,*** String ***NomTable*****);**

Où SchemaType peut prendre une des valeurs suivantes :

- **Mapped** ➔ Applique le mappage sur les tables déjà existantes du DataSet
- **Source** ➔ Pas de modification sur les tables existantes

Les informations de schéma récupérées pour les colonnes sont

- AllowDBNull
- AutoIncrement
- MaxLength
- ReadOnly
- Unique

Si la ou les colonnes définissant la clé primaire sont renvoyées par la requête, la clé est ajoutée au schéma. Sinon la clé primaire sera définie par une colonne ayant la contrainte d'unicité s'il y en a une.

La clé primaire et les contraintes d'unicité sont les seules contraintes récupérées.

GetFillParameters

Permet de récupérer les paramètres de l'utilisateur dans la requête SELECT.

Update

C'est la méthode déclenchant l'envoie des modifications faites dans le DataSet vers la source de données. Il existe plusieurs versions surchargées mais la plus commune est :

public override int Update**(**DataSet **NomDataSet,** string ***NomDataTable*****)**

La valeur retournée correspond au nombre de lignes correctement modifiées dans la source.

Je vous ai déjà expliqué le fonctionnement général mais nous allons recommencer car il est primordial de bien comprendre ce qui se passe.

Votre DataSet contient la DataTable qui va servir de source de modification. Celle-ci peut être vue comme une collection d'objets DataRow. Lors de l'appel de la méthode Update, le DataAdapter va regarder la propriété RowState de chaque ligne et tenter de répercuter les modifications de toutes celles qui ne sont pas marquées 'Unchanged'. Normalement la lecture se fait dans l'ordre de tri actuellement sélectionné mais ceci peut être changé par l'appel de la méthode GetChanges du DataSet. Ensuite pour chaque ligne la séquence suivante va s'exécuter :

1. Vérification des informations de mappage ;
2. Levée de l'événement OnRowUpdating ;
3. Sélection de l'objet Command approprié. Si celui-ci n'est pas défini et s'il existe un objet CommandBuilder, ce dernier est appelé ;
4. Ecriture et envoie de la commande ;
5. Selon les spécifications de la commande, la ligne peut être mise à jour dans le Dataset et il peut y avoir récupération de paramètres de sortie / retour ;
6. Levée de l'événement OnRowUpdated ;
7. Appel de la méthode AcceptChanges .

Cette opération n'est pas une opération par lot. C'est à vous de déclarer une transaction si vous voulez que la mise à jour soit atomique.

A chaque ligne une exception peut se déclencher si :

- Les informations de mappage ne permettent pas d'identifier la cible.
- Il y a concurrence optimiste.

Si une erreur se produit vous pouvez suivre trois stratégies.

- Annuler toute l'opération si vous êtes en mode transactionnel.
- Traiter l'erreur à la volée en la gérant dans l'événement OnRowUpdated.
- Ignorer les erreurs en mettant la propriété ContinueUpdateOnError à vraie. Dans ce cas, il y aura remontée des valeurs d'erreur vers la propriété RowError de la ligne.

Comme nous le voyons, une mise à jour demande le paramétrage correct d'un grand nombre d'éléments. Il est donc indispensable de bien définir la stratégie que l'on veut suivre, et recommandé d'utiliser un DataAdapter par objet DataTable.

Evènements

RowUpdating

Se produit avant l'exécution de la commande sur la source. Les paramètres de l'événement sont :

Propriété	Description
Command	Obtient ou définit SqlCommand à exécuter durant Update.
Errors	Obtient les erreurs générées par le fournisseur de données .NET Framework lors de l'exécution de Command.
Row	Obtient le DataRow à envoyer par l'intermédiaire de Update.
StatementType	Obtient le type de l'instruction SQL à exécuter.
Status	Obtient UpdateStatus de la propriété Command.
TableMapping	Obtient le DataTableMapping à envoyer par l'intermédiaire de Update.

Cet événement sert à modifier le traitement d'une ou plusieurs lignes avant leur traitement.

RowUpdated

Déclenché après une tentative de modification, quel qu'en soit le résultat. Les arguments de l'événement sont :

Propriété	Description
Command	Obtient ou définit SqlCommand qui est exécuté lorsque Update est appelé.
Errors	Obtient les erreurs générées par le fournisseur de données .NET Framework lors de l'exécution de Command.
RecordsAffected	Obtient le nombre de lignes modifiées, insérées ou supprimées par l'exécution de l'instruction SQL.
Row	Obtient le DataRow envoyé par l'intermédiaire de Update.
StatementType	Obtient le type de l'instruction SQL exécutée.
Status	Obtient UpdateStatus de la propriété Command.
TableMapping	Obtient le DataTableMapping envoyé par l'intermédiaire de Update.

Cet événement sert généralement au traitement des erreurs.

FillError

Se produit lors d'un problème de remplissage. C'est généralement soit une conversion de type entraînant une perte d'information soit un non-respect de contrainte. Les arguments sont :

Propriété	Description
Continue	Obtient ou définit une valeur indiquant s'il convient de poursuivre l'opération de remplissage malgré l'erreur.
DataTable	Obtient le DataTable en cours de mise à jour au moment où l'erreur s'est produite.
Errors	Obtient les erreurs gérées actuellement.
Values	Obtient les valeurs de la ligne en cours de mise à jour au moment où l'erreur s'est produite.

Attention : Les erreurs du SGBD ne lèvent pas cette exception.

Commandes du DataAdapter

Nous allons voir ici comment définir et paramétrer les objets Command du DataAdapter. Dans cette partie nous n'allons pas revoir la syntaxe qui est la même que celle de l'objet Command, mais nous discuterons de stratégies d'action. Les objets Command du DataAdapter se manipulent par l'intermédiaire de la propriété du même nom de l'objet DataAdapter.

SelectCommand

Elle doit être toujours définie. Si la commande ne renvoie pas de ligne, aucune exception ne sera levée. La composition de cette commande est très importante. En effet, selon sa composition, la création des clés dans le Dataset peut être différente. En cas de besoin, le DataAdapter cherche toujours à créer une clé primaire sur la table. Si vous utilisez le CommandBuilder, il utilisera les informations de cette commande pour générer le code de mise à jour.

La classe CommandBuilder

A la différence d'ADO, le DataAdapter ne génère pas spontanément le code de mise à jour. Il attend que vous lui fournissiez les objets Command nécessaire à la mise à jour dans ses objets UpdateCommand, DeleteCommand et InsertCommand. Toutefois vous pouvez vous rapprocher des fonctionnalités ADO en déclarant plutôt un objet CommandBuilder pour qu'il prenne en charge le code de mise à jour. La génération du code se fait grâce aux informations récupérables par le code de l'objet SelectCommand. Si vous modifiez celui-ci, vous devez appeler la méthode RefreshSchema du CommandBuilder. Faute de cela, les anciennes commandes continueront d'être utilisée.

L'utilisation de cet objet est restreinte au cas où votre requête ne concerne qu'une table **unique**.

Propriétés

(Oledb, sql)DataAdapter

Obtient ou renvoie le DataAdapter lié au CommandBuilder.

QuotePrefix, QuoteSuffix (String)

Permet de définir les caractères d'encadrement à utiliser lorsque les noms d'objets contiennent des caractères interdits (espace par exemple). La définition de ces propriétés permet au CommandBuilder de générer un code relativement robuste.

Méthodes

DeriveParameters

Permet d'affecter les paramètres d'une commande vers les commandes du CommandBuilder.

GetDeleteCommand, GetInsertCommand, GetUpdateCommand

Renvoie l'objet commande spécifié utilisé par le CommandBuilder.

RefreshSchema

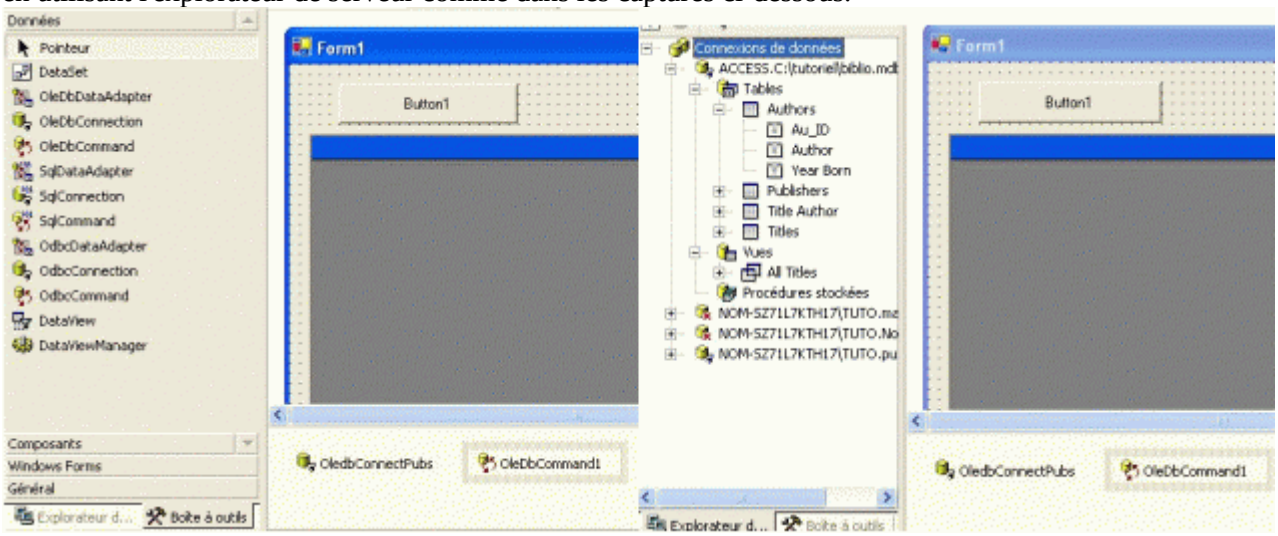
Régénère les commandes actions en relisant la commande SELECT.

Discussions et exemples de codes

Utilisation de l'IDE

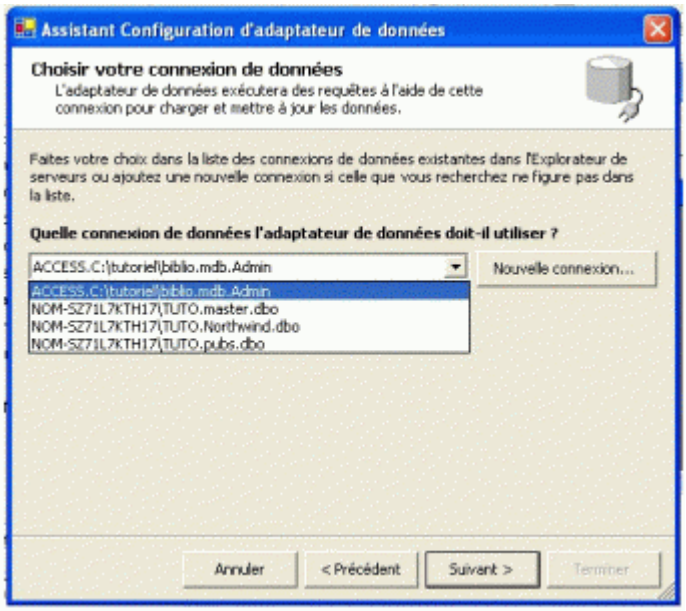
Le présent chapitre ne va parler que de l'utilisation des composants de données dans l'interface de développement. Ces composants se trouvent dans la boite à outils sous l'onglet "Données".

A la différence de VS6, les composants non visuels ne s'ajoutent pas sur la feuille mais sur une barre qui se situe sous la fenêtre. A partir de là, il me suffit de les sélectionner pour pouvoir accéder à leur fenêtre de propriétés. Dans le cas des composants de données je peux les sélectionner en partant de la boite à outils ou en utilisant l'explorateur de serveur comme dans les captures ci-dessous.

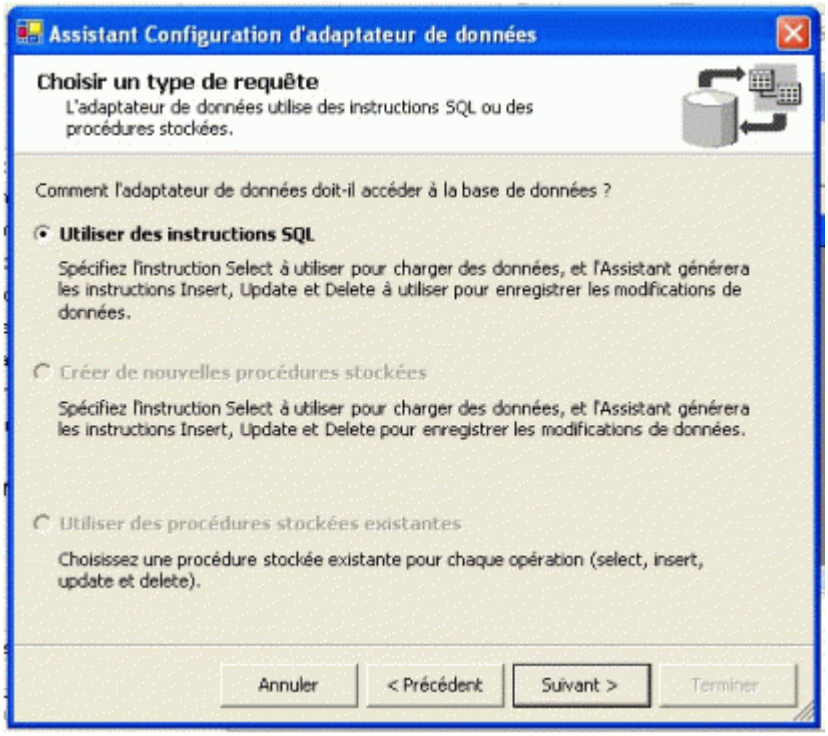


Les composants DataAdapter vous proposent l'utilisation d'un assistant lors de leur ajout.

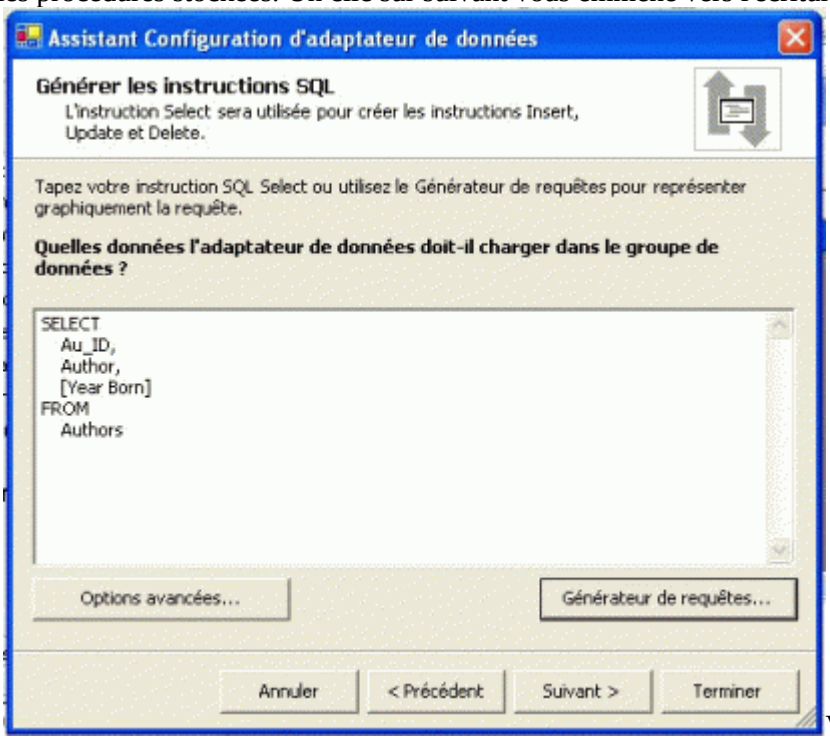
Assistant du DataAdapter



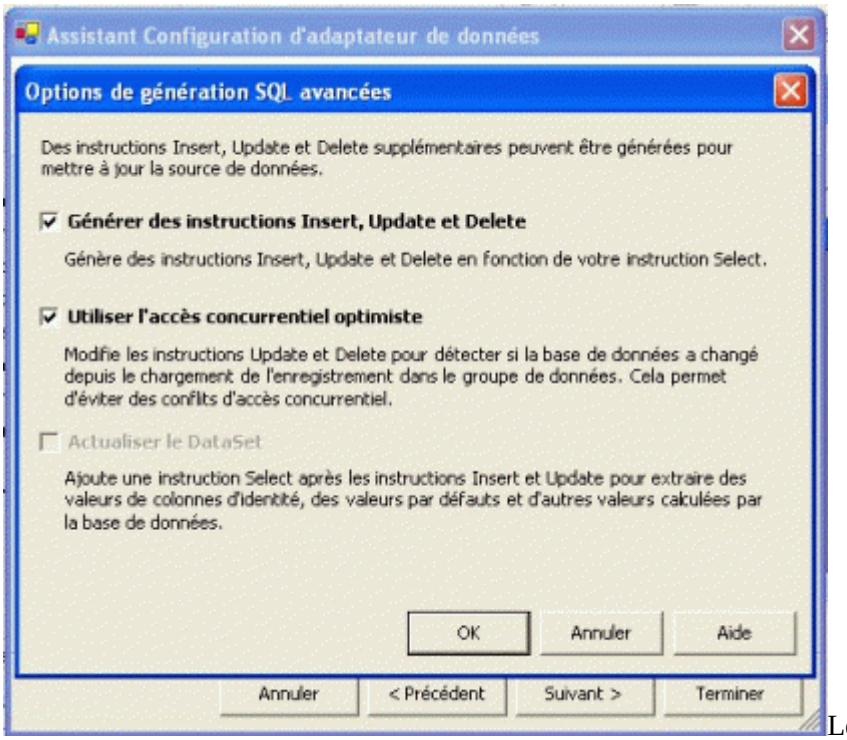
Le premier écran vous demande juste la connexion qu'il faut utiliser dans un Combo classique. La liste reprise est celle des sources visibles dans l'explorateur de serveur. Une fois celle-ci choisie, vous cliquez sur suivant :



Dans cet écran, vous avez le choix entre l'utilisation de requête SQL ou le travail avec des procédures stockées existantes ou à créer. Dans mon cas le choix n'existe pas puisque ma source de données n'accepte pas les procédures stockées. Un clic sur suivant vous emmène vers l'écriture de la requête SELECT.



Vous pouvez utiliser le générateur de requête si vous le souhaitez, pour ma part je préfère l'utilisation de mes gros doigts boudinés. En cliquant sur options avancées vous obtiendrez la boite suivante :



Les choix proposés ici sont suffisamment importants pour que nous les expliquions ici.

Générer des instructions Insert, Update et Delete

Ce sujet va beaucoup revenir dans cette partie. Tout le choix proposé consiste à laisser l'assistant générer les commandes de mises à jour en fonction de votre instruction SELECT, c'est à dire dans une certaine mesure se comporter comme un objet CommandBuilder, soit vous laissez les gérer vous-mêmes. La génération des commandes par l'assistant est d'assez bonne qualité, aussi dans les cas simples je vous conseille de l'utiliser. Nous verrons plus loin quand il convient de reprendre la main.

Utiliser l'accès concurrentiel optimiste

Cette phrase est un non-sens. En fait il s'agit de définir si vous voulez que tout les champs d'origines apparaissent dans la clause WHERE ou si vous ne voulez utiliser que les champs clés (scénario last-in win).

Actualiser le Dataset

Si la source supporte les lots de requêtes, il y aura ajout d'une requête SELECT après les requêtes INSERT ou UPDATE pour vous aider à gérer la mise à jour.

Enfin un dernier clic sur suivant vous emmènera vers un rapport signalant les réussites et les échecs rencontrés par l'assistant.

Composant Dataset

Quand bien même avez-vous créé votre DataAdapter, l'ajout d'un composant Dataset sur votre feuille ne liera pas les deux. Lorsque vous ajoutez le composant Dataset il vous demande juste si vous souhaitez créer un Dataset typé, ce qui sous-entend que la classe existe déjà dans votre projet ou un Dataset Non-typé. Il n'existe pas de solution pour lier le composant Dataset et le composant DataAdapter en dehors du code. Notez que de toute façon, vous aurez besoin du code pour remplir le Dataset.

Une erreur commune consiste à utiliser le DataAdapter et à choisir 'Générer le groupe de données'. On voit alors apparaître un composant Dataset dans la fenêtre mais celui-ci est un Dataset **fortement typé**.

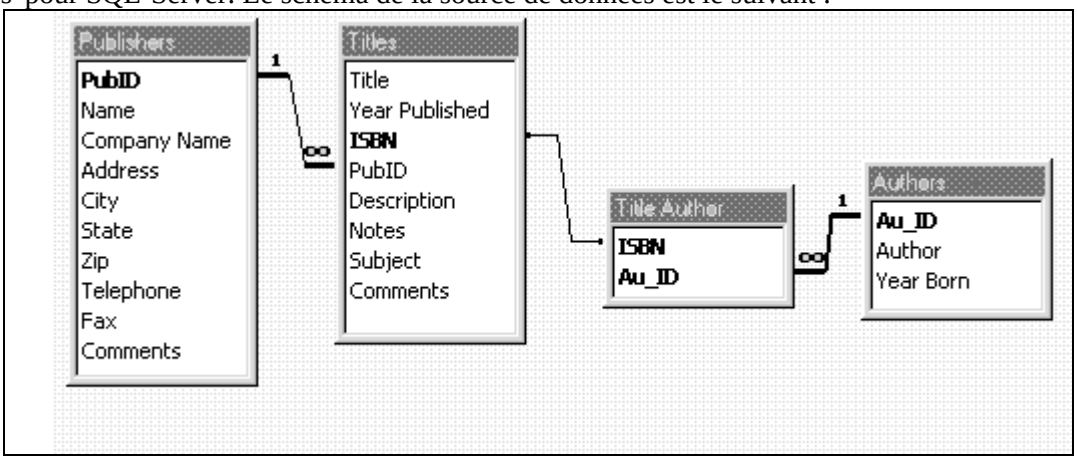
Votre composant Dataset peut commencer à être manipulé à la création. Vous pouvez lui ajouter des tables et des relations en manipulant ses propriétés. Vous pouvez aussi ajouter des composants DataView et les configurer à la création.

Pour savoir si votre Dataset est typé ou non il vous suffit de regarder dans l'explorateur de solution. Si vous voyez apparaître un fichier '.xsd' et que celui-ci contient un fichier '.cs', vous êtes en présence d'un Dataset fortement typé.

Concepts généraux : Créer un Dataset

Pour obtenir une bonne représentation des objets DataAdapter et Dataset il n'est pas possible de les dissocier. Il est tout à fait possible de créer un Dataset de toute pièce, de lui ajouter des contraintes et des relations, de ne le mettre jamais en contact avec une source de données existantes. Ici nous allons plutôt discuter de l'échange d'informations entre un Dataset et une source de données ; donc de l'utilisation d'un Dataset et d'un DataAdapter.

Dans ce chapitre, nous allons voir comment créer un Dataset représentant la base Access 'Biblio' ou 'Pubs' pour SQL-Server. Le schéma de la source de données est le suivant :



J'utilise cet exemple, car la base est assez simple, il y a peu de relations et peu de contraintes. Détaillons maintenant les contraintes et validation des tables.

Publishers

PubId ⇒ Clé primaire ; Int32 ; AutoIncrement; Indexé.

Titles

Title ⇒ Indéxé, Null interdit

ISBN ⇒ Clé primaire ; Texte (20) ; Indexé

PubId ⇒ Clé étrangère ; Int32

Authors

Au_Id ⇒ Clé primaire ; Int32 ; AutoIncrement; Indexé.

Author ⇒ Null interdit

Title Author

ISBN ⇒ Clé étrangère ; Texte (20)

Au_ID ⇒ Clé étrangère ; Int32

La clé primaire de la table est composée de ces deux champs

Il existe trois relations dans cette base (autant que de clé étrangères).

Discussion autour du schéma

Lorsqu'on aborde un programme avec ADO.NET, c'est une des premières questions à mettre dans la balance, récupérer le schéma ou non. Lorsqu'on utilise ADO, la question ne se pose pas puisque le recordset récupère le schéma, même quand on n'a pas besoin d'ailleurs, mais dans notre cas il s'agit de deux cheminements possibles. La récupération du schéma est une opération coûteuse en ressources et selon la méthode employée, le retour sera sensiblement différent. Les éléments de réponse se trouvent souvent dans l'exposé du problème.

- Je ne connais rien de la source lorsque j'écris mon programme. La récupération dynamique du schéma est obligatoire.
- Je connais ma source de données
  - Je veux pouvoir la manipuler comme une entité compacte, j'utilise un Dataset fortement typé
  - Je veux un travail plus souple, modification des contraintes, modification de structure, etc je gère par le code

Le premier cas et le second sont clairs mais le troisième est un peu plus complexe. En effet je peux souhaiter alléger le développement et utiliser des récupérations partielles du schéma par le code. Elles seront généralement partielles car la récupération du schéma complet n'est pas une opération évidente.

Lorsque je procède à une opération de remplissage par mon DataAdapter, seule les données sont récupérées. Si je veux récupérer la clé primaire, je dois affecter AddWithKey à la propriété MissingSchemaAction du DataAdapter avant d'appeler Fill. Si je veux récupérer d'autres informations en plus de la clé, je dois faire un appel FillSchema avant le Fill. Je récupère ainsi les colonnes auto-incrémentées et les contraintes verticales. Néanmoins je ne reçois jamais les informations de clés étrangères.

N.B : Il n'y a jamais import des index dans le schéma. ADO.NET ne gère pas les index dans le sens SGBD du terme puisqu'il n'y a jamais recherche d'une valeur par un objet ADO.NET directement sur la source de données. Par contre les objets DataView gère des index sur les Dataset.

Je sais qu'il peut être tentant de faire de la récupération dynamique, mais j'avoue que pour ma part j'essaye de faire le maximum à la conception pour alléger l'exécution. Voyons maintenant trois exemples montrant le fonctionnement de ces trois cas.

Récupérer le schéma

Dans cet exemple, nous allons récupérer les informations de schéma par le code, partant du principe que nous ne connaissons que le nom de la source. L'objet OleDbConnection fournit une méthode GetOleDbSchemaTable qui permet de récupérer des informations de schéma. Elle est de la forme suivante :

public DataTable GetOleDbSchemaTable(Guid *schema***, object[]** *restrictions***)**

```
    this.Button1.Click += new System.EventHandler(Button1_Click);
    …
    private void Button1_Click(object sender As, System.EventArgs e) {
        string strConn, strSQL;
        strConn = @"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;";
        OleDbConnection MaConn = new OleDbConnection(strConn);
        MaConn.Open();
        DataTable MaTable =
MaConn.GetOleDbSchemaTable(OleDbSchemaGuid.Tables, new object[] {null,
null, null, "TABLE"});
        string NomTable;
        DataSet MonDataset = new DataSet();
        foreach(DataRow MaLigne in MaTable.Rows) {
            NomTable = MaLigne.Item("TABLE_NAME");
            OleDbDataAdapter TabDtaAdapt = new OleDbDataAdapter("SELECT *
FROM [" & NomTable & "]", MaConn);
            TabDtaAdapt.Fill(MonDataset, NomTable);
            TabDtaAdapt.FillSchema(MonDataset, SchemaType.Mapped,
NomTable);
            TabDtaAdapt.Dispose();
        }
        MonDataset.WriteXmlSchema(@"c:\\tutoriel\\dotnet\\type.xsd");
    }
```

J'ai avec ce code simple récupéré l'ensemble de mes tables avec leurs clés primaires et leurs contraintes de colonnes. En utilisant la méthode WriteXmlSchema je peux donner une certaine persistance à mon modèle. Il nous faut cependant encore ajouter les relations. Pour cela nous allons ré interroger le schéma de la base.

```
MaTable.Clear() ;
//interrogation de la table des relations
MaTable = MaConn.GetOleDbSchemaTable(OleDbSchemaGuid.Foreign_Keys, new
object[] {null, null, null, null, null, null});
DataTable TableParent, TableEnfant; DataColumn ColParent, ColEnfant; Dim
ForeignKeyConstraint EtrKey; DataRelation MaRelat;
foreach(MaLigne in MaTable.Rows) {
    TableParent = MonDataset.Tables(MaLigne.Item("Pk_Table_Name"));
    TableEnfant = MonDataset.Tables(MaLigne.Item("Fk_Table_Name"));
    ColParent = TableParent.Columns(MaLigne.Item("Pk_Column_Name"));
    ColEnfant = TableEnfant.Columns(MaLigne.Item("Fk_Column_Name"));
    EtrKey = new ForeignKeyConstraint(ColParent, ColEnfant);
    if (MaLigne.Item("DELETE_RULE").IndexOf("NO ACTION") > 0)
        EtrKey.DeleteRule = Rule.None;
    else
        EtrKey.DeleteRule = Rule.Cascade;
    if (MaLigne.Item("UPDATE_RULE").IndexOf("NO ACTION") > 0)
        EtrKey.UpdateRule = Rule.None;
    else
        EtrKey.UpdateRule = Rule.Cascade;
    if (EtrKey.UpdateRule == Rule.Cascade && EtrKey.DeleteRule ==
Rule.Cascade)
        EtrKey.AcceptRejectRule = AcceptRejectRule.Cascade;
    EtrKey.ConstraintName = TableEnfant.TableName & "_fk_" &
TableParent.TableName;
    TableEnfant.Constraints.Add(EtrKey) ;
    MaRelat = new DataRelation(TableEnfant.TableName & "_rel_" &
TableParent.TableName, ColParent, ColEnfant, true) ;
    MonDataset.Relations.Add(MaRelat) ;
}
MonDataset.WriteXmlSchema(@"c:\\tutoriel\\dotnet\\type.xsd") ;
```

Il serait un peu différent pour le fournisseur managé SQL. En effet celui-ci ne met pas à votre disposition une méthode de récupération de schéma identique à GetOleDbSchemaTable. Pour récupérer le schéma vous devez utiliser des vues de schéma avec un objet Command ou avec des requêtes de schéma. Par exemple pour récupérer la liste des tables :

```
SqlClient.SqlDataAdapter schemaDA = new SqlClient.SqlDataAdapter("SELECT
* FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_TYPE = 'BASE TABLE' ORDER BY
TABLE_TYPE", nwindConn);
DataTable schemaTable = new DataTable();
schemaDA.Fill(schemaTable) ;
```

Je peux donc assez aisément reconstruire ma base de données dans un dataset sans connaître la structure à l'origine. Ce type de cas est toutefois très rare. Heureusement car le coût en ressources est lourd. Pourtant j'ai géré au mieux le code pour privilégier sa rapidité et pour cela j'ai fait une impasse importante.

Tel que nous le voyons, notre Dataset nouvellement créé n'a pas possibilités de transmission de ces modifications vers la source sauf à générer un code extrêmement lourd.

Pour générer correctement ces modifications, un dataset utilise généralement un DataAdapter par table. Dans mon cas je n'ai qu'un seul DataAdapter. Il serait techniquement possible de gérer l'ensemble des modifications avec un seul DataAdapter.

```
OleDbDataAdapter MonDtaAdRetour = new OleDbDataAdapter();
OleDbCommand[] TabSelectCommand = new OleDbCommand
[MonDataset.Tables.Count];
OleDbConnection MaConn = new
OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;");
OleDbCommandBuilder CmdBuild = new OleDbCommandBuilder(MonDtaAdRetour);
CmdBuild.QuotePrefix = "[";
CmdBuild.QuoteSuffix = "]";
for(int cmpt = 0 ; cmpt <= TabSelectCommand.GetUpperBound(0) ; cmpt++){
    TabSelectCommand(cmpt) = new OleDbCommand("SELECT * FROM [" &
MonDataset.Tables(cmpt).TableName & "]", MaConn);
}
if(MonDataset.HasChanges) {
    foreach(DataTable MaTable in MonDataset.Tables) {
        If(MaTable.GetChanges.Rows.Count > 0) {
            MonDtaAdRetour.SelectCommand =
TabSelectCommand(MonDataset.Tables.IndexOf(MaTable));
            CmdBuild.RefreshSchema();
            MonDtaAdRetour.Update(MaTable) ;
        }
    }
}
```

Vu comme cela ce n'a pas l'air trop compliqué. Pourtant ce code ne fonctionnera pas bien. Dans le cas de table en relation, l'ordre des modifications n'est pas sans importance. Supposons que j'ajoute et que je supprime des éditeurs et des titres. Je dois forcément insérer les éditeurs avant les titres et supprimer les titres avant les éditeurs. Mon code est bien loin de faire cela et une telle modification l'alourdirait considérablement. Je pourrais prendre d'autres méthodes, comme une collection de DataAdapter, mais toutes serait difficiles à gérer correctement.

Néanmoins, il est possible de gérer cela au prix d'un ralentissement du code. Nous allons maintenant regarder le cas plus fréquent ou je connais la source.

Dataset fortement typé

Comme je vous l'ai déjà dit, il s'agit d'un fichier de schéma XSD et d'un module de classe. Nous allons créer celui-ci presque uniquement avec l'environnement intégré.

Mon but étant sensiblement le même que précédemment, je vais recréer ma source comme un Dataset fortement typé. Pour mémoire, ceci consiste à recréer un schéma xsd de ma source et à générer un module de classe liée à celui-ci. Un Dataset fortement typé permet un codage plus simple et un accès à de très bonnes performances, mais il est lié à une source prédéfinie.

Pour ma part, je vous recommande chaudement l'utilisation des Dataset fortement typés chaque fois que vous travaillez sur une source unique et connue.

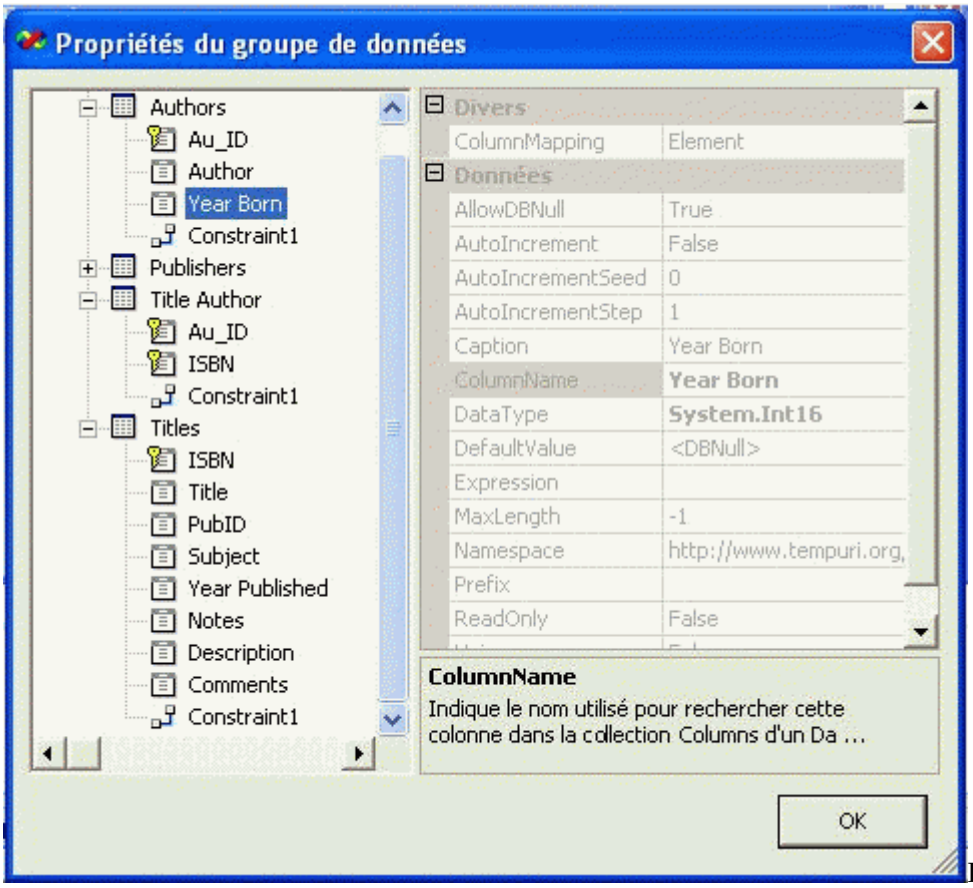
Je vais donc ajouter à ma feuille un composant connexion (OleDbCnnPubs) et quatre composants OleDbDataAdapter, un par table.

- OleDbDAAuthors pour la table Authors
- OleDbDAPublishers pour la table Publisher
- OleDbDATitles pour la table Titles
- OleDbDATitAut pour la table Title Author

J'ai rangé mes requêtes SQL dans un ordre particulier qui correspondait à mes souhaits ultérieurs, mes nous verrons que ce n'est utile que pour les DataView.

Pour créer mon Dataset fortement typé, je vais faire un clic droit sur un de mes DataAdapter et choisir 'Générer le groupe de données'. Je vais sélectionner toutes les tables et ajouter le composant ainsi créé dans le concepteur. J'ai nommé mon Dataset 'dtsPubs' mais le composant ajouté s'appelle 'dtsPubs1'. Cela vient du fait qu'il existe maintenant un fichier 'dtsPubs.xsd' dans l'explorateur de solution. Si je choisis d'afficher tous les fichiers dans celui-ci, je vois apparaître un fichier 'dtsPubs.cs' qui est mon fichier de classe. Celui-ci étant extrêmement long je ne vais pas vous le recopier ici.

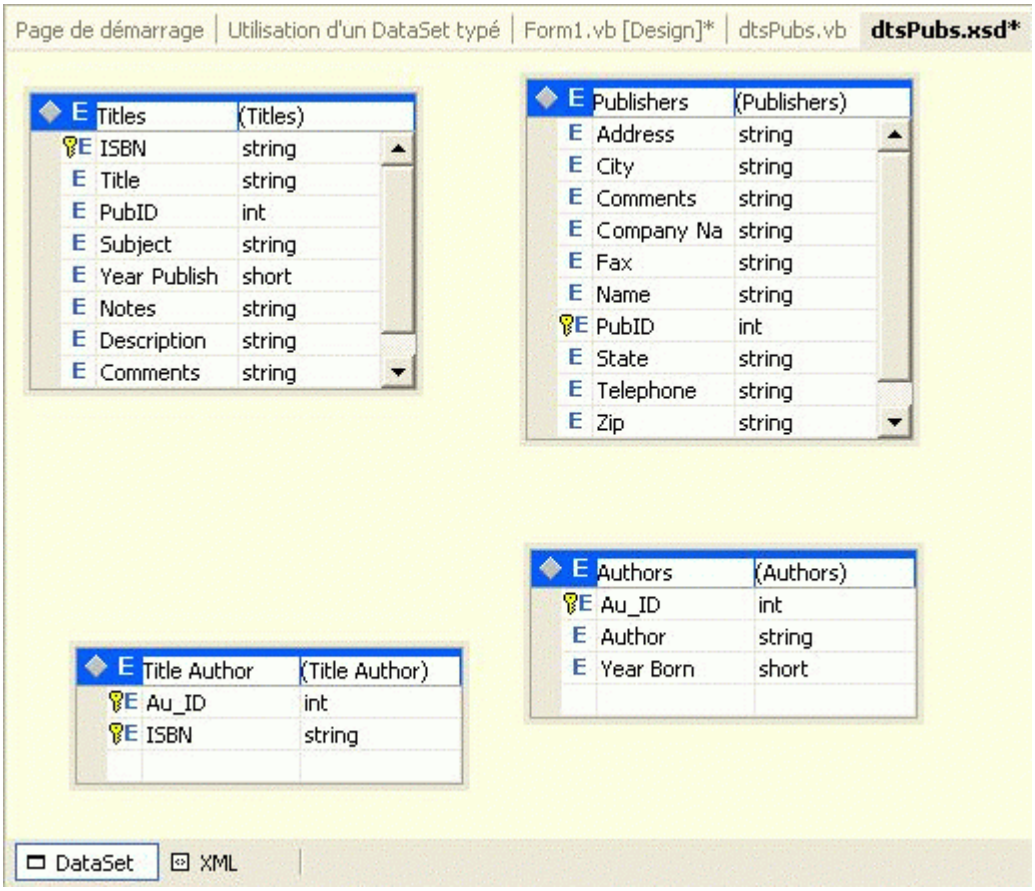
Si je clique droit sur mon composant Dataset je peux choisir d'aller dans les propriétés du groupe de données. J'aurais alors l'écran suivant :



De cette boite, je peux facilement visualiser mon schéma, mais pas le modifier. Mais dans ce groupe de données, pas de relations.

Ajouter les relations

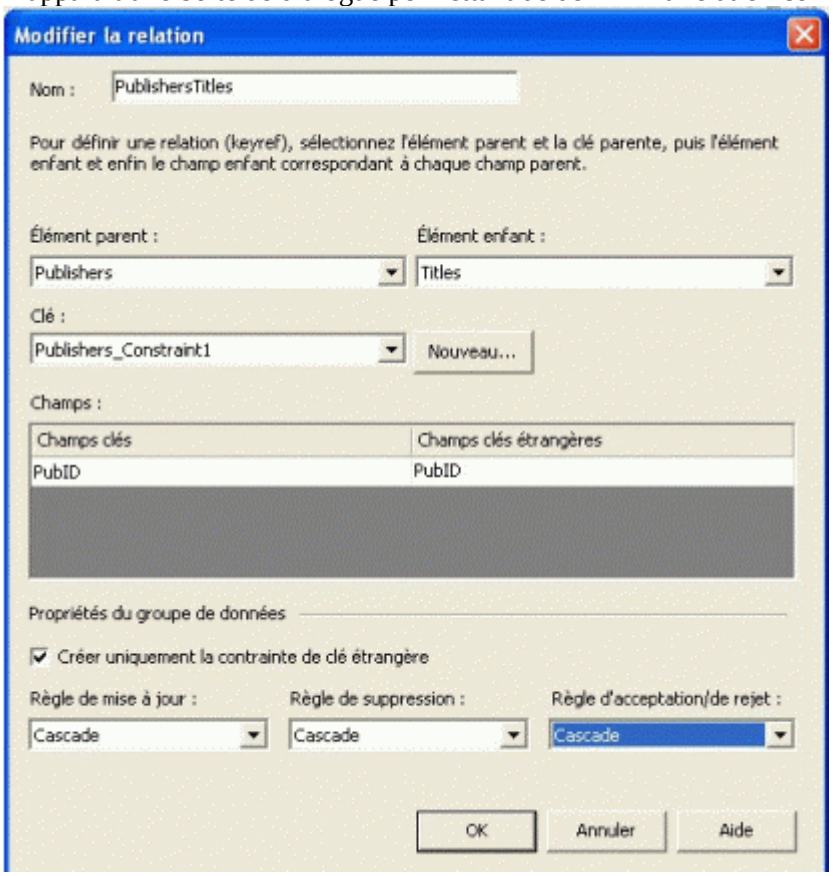
Un autre clic droit sur mon Dataset et je choisis 'Afficher le schéma'. Je vois alors apparaître l'écran suivant.



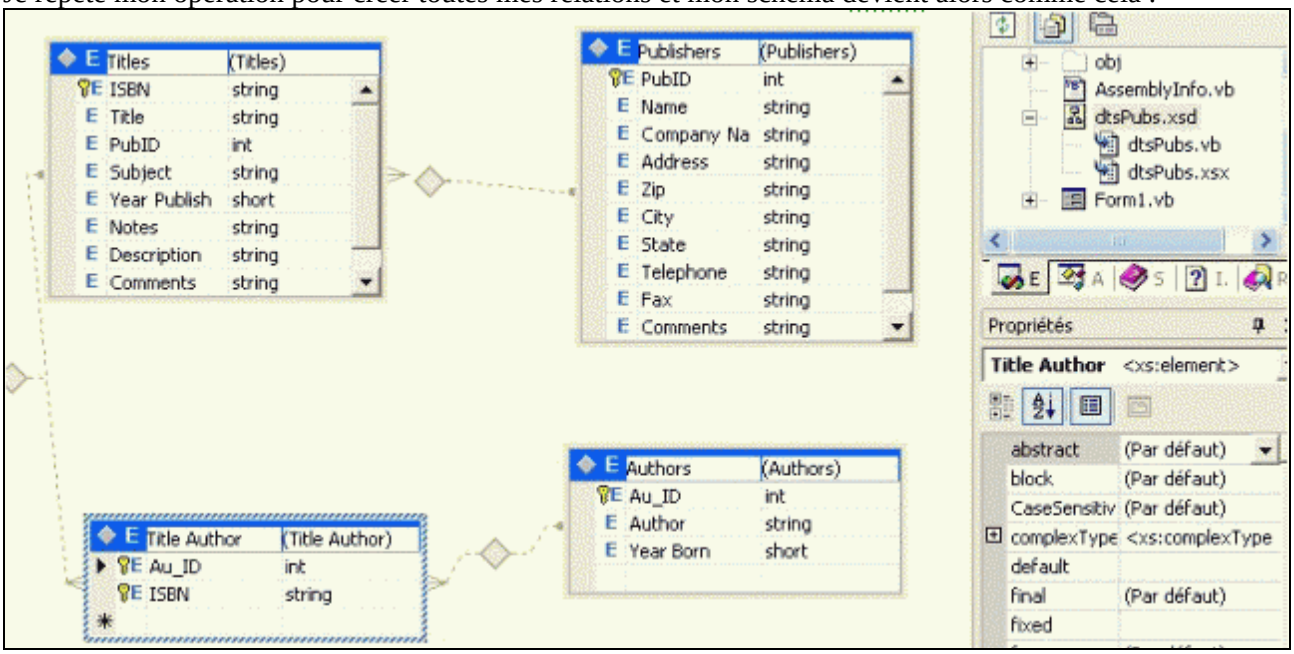
Vous voyez que l'onglet du bas me permet de basculer la vision entre Dataset et XML. Le Dataset et le fichier XSD sont synchronisés. Dans cette vue je peux modifier mon schéma, mon fichier XSD se modifiera automatiquement, et réciproquement. Faites très attention dans cette vue, vous pouvez modifier le schéma sans problème. Malheureusement les problèmes risquent de se passer à l'exécution si vous n'êtes pas particulièrement vigilant.

Dans mon schéma, je vais sélectionner une des tables à mettre en relation puis sélectionner dans le menu 'Schema' – 'Ajouter une relation'.

Il apparaît une boite de dialogue permettant de définir ma relation comme je le souhaite.



Je répète mon opération pour créer toutes mes relations et mon schéma devient alors comme cela :



Les relations sont maintenant écrites dans le fichier XSD.

Vous allez peut être vous dire que c'était un peu lourd mais vous possédez maintenant un Dataset très facile à utiliser. Toutefois, si vous l'utilisez comme ça il ne contient aucune donnée. Un peu de code va devoir être nécessaire pour remplir le Dataset.

```
this.Form1.Load += new System.EventHandler(Form1_Load);
…
private void Form1_Load(object sender, System.EventArgs e) {
    this.DtsPubs1.EnforceConstraints = false;
    this.OleDbDAAuthors.Fill(this.DtsPubs1.Authors);
    this.OleDbDAPublishers.Fill(this.DtsPubs1.Publishers);
    this.OleDbDATitAut.Fill(this.DtsPubs1.Title_Author);
    this.OleDbDATitles.Fill(this.DtsPubs1.Titles);
    this.DtsPubs1.EnforceConstraints = true ;
    //démo de codage fortement typé
    dtsPubs.PublishersRow NouvLigne = this.DtsPubs1.Publishers.Rows(1);
    if(NouvLigne.IsAddressNull) NouvLigne.Address = "chez moi";
}
```

Je n'ai pas de démonstration fantastique de la différence de codage, mais essayez-le une fois et vous verrez une nette différence.

Stratégies de mise à jour

Nous allons traiter ici de la partie la plus complexe de l'utilisation d'ADO.NET. Je ne vais traiter ici que de l'utilisation d'un DataAdapter.

Il existe globalement trois choix pour gérer les opérations de mise à jour

- Le CommandBuilder
- La génération par l'assistant
- La génération par le développeur

Contrairement à ce que vous pouvez penser, il y a de nombreux cas ou l'utilisation d'un CommandBuilder ou de l'assistant suffit largement à une bonne gestion de la mise à jour.

Rappels SQL

Pour modifier les **données** de la source, on utilise des requêtes actions. Une requête action ne doit porter que sur une seule table. Toutes les requêtes actions doivent respecter l'ensemble des contraintes et restrictions pour être valides. On en distingue trois.

Ajout avec INSERT…INTO

```
INSERT [INTO] nom_de_la_table [(liste_des_colonnes)]
{VALUES (liste_des_valeurs)}
```

Cette requête ne présente pas de problème d'identification de ligne puisqu'il s'agit d'une nouvelle ligne, il convient de tempérer ce jugement lorsqu'on travaille avec des clés générées. Les valeurs omises lors de la requête prennent la valeur par défaut si celle-ci existe, la valeur NULL sinon. Il est possible de ne pas spécifier la liste des colonnes, mais ceci sous-tend que la liste des valeurs correspond à toutes les colonnes. Très honnêtement je vous déconseille vivement d'omettre la liste des colonnes car cela demande de savoir dans quel ordre mettre les valeurs pour qu'il corresponde à l'ordre des colonnes visées.

Lorsque la table ciblée possède des colonnes auto-incrémentées, on ne passe pas de valeurs dans la les colonnes et on ne met pas non plus le nom de ces colonnes dans la liste des colonnes.

Suppression avec DELETE

```
DELETE [FROM] nom_de_la_table
[WHERE condition]
```

Dans le contexte de concurrence que nous allons voir ci-après, c'est de loin la requête la plus dangereuse. La condition WHERE doit évidemment permettre d'identifier la ou les lignes cibles. Cette requête ne présente pas de difficulté majeure.

Modification avec UPDATE

```
UPDATE nom_de_la_table e
SET colonne = valeur [, colonne2 = valeur2 ...]
[WHERE condition]
```

C'est autour de cette requête que vont se concentrer la plupart des difficultés. Dans le concept, on passe dans la clause SET la liste des nouvelles valeurs pour les colonnes ciblées, et on identifie la ou les lignes par la clause WHERE. Les colonnes non ciblées sont laissées dans leur état d'origine.

La première chose que nous allons regarder est la construction du critère WHERE.

Choisir les restrictions du critère

Retournons dans notre vision ADO.NET. Les requêtes utilisées par le DataAdapter doivent pouvoir toujours identifier la ligne cible de façon unique, ceci ne concernant évidemment pas les requêtes d'ajout. Cette unicité d'identification est induite par le DataAdapter puisque celui-ci examine les lignes successivement et envoie les requêtes de misa à jour **successivement**. Pour identifier une ligne de façon unique, on utilise la clé primaire de la table. Une table sans clé primaire et ou aucun critère d'unicité ne peut être défini est inexploitable.

Cette condition quoique nécessaire, n'est pas suffisante. Comme je vous l'ai déjà dit, ADO.NET travaille toujours dans un contexte de concurrence optimiste. Ceci veut dire qu'il vous appartiendra de vérifier que la ligne que vous ciblez n'a pas déjà été modifiée par un autre utilisateur. Il existe dès lors plusieurs scénarii possibles.

La clé uniquement

Connu sous le nom de 'last-in Win' (le dernier est vainqueur). Comme votre ligne n'est identifiée que par sa valeur de clé, vos modifications écraseront toutes les modifications des autres utilisateurs sans que vous soyez averti de la modification de la ligne. Je n'ai jamais vu d'utilisation fondée de ce genre de technique dans un contexte multi-utilisateurs.

La clé et la version

De nombreux SGBD gère la notion de numéro de version. On la trouve parfois sous le nom de TimeStamp du fait que certains SGBD donnent une valeur de type Date/heure pour gérer la version. Il faut évidemment que votre table soit conçue avec cette information. Dans le fonctionnement, chaque modification de la ligne dans la source entraîne l'attribution d'une nouvelle valeur de version, forcément unique, à la ligne.

Lorsque vous rapatriez une ligne vers votre Dataset, elle contient son numéro de version. Si vous renvoyez cette ligne pour mise à jour, son numéro de version est comparé avec celui existant dans la source et la modification n'est effective que si ceux-ci sont identiques.

Ce principe de fonctionnement est très fiable, mais tous les SGBD ne le gèrent pas et il faut que le schéma soit conçu avec. Si c'est vous qui concevez la source, utilisez le dès que possible.

Notez que le CommandBuilder ne gère pas les numéros de version dans sa logique.

La clé et les colonnes modifiées

C'est la logique du moteur de curseur ADO. Celle-ci est une restriction partielle de l'identification. La clause WHERE contient la clé et les valeurs d'origines de toutes les colonnes modifiées. Ceci implique que pour identifier la ligne, il faut qu'aucune des colonnes que vous voulez modifier ne l'ai été par un autre utilisateur. Par contre, si un autre champ a été modifié cela ne pose pas de problème. Cette technique est très performante en termes de modifications successives, mais elle n'est pas gérée par ADO.NET. Dans sa stratégie de mise à jour, ADO.NET se base sur des objets Command défini une fois. Or cette technique sous-tend que la requête soit créée à la volée. Il est évidemment possible de la reproduire par le code, mais il convient alors de peser le coût d'exécution par rapport aux avantages espérés.

La clé et tous les champs

C'est le cas que vous trouverez défini dans la documentation comme 'concurrence optimiste' (sic)

Cela revient à mettre les valeurs d'origine de tous les champs dans la clause WHERE. Si au moins une valeur a été modifiée par un autre utilisateur, la modification ne pourra avoir lieu. Cela revient globalement à la même chose que d'utiliser un numéro de version, mais en plus lourd. Toutefois tous les SGBD supportent ce type de requête. Gardez à l'esprit que l'assistant comme le CommandBuilder ne vous proposent que cette option ou le scénario 'last-in Win'.

Etude des cas particuliers

Delete or not delete : That is…

Je suis toujours surpris lorsque je vois autre chose que la clé dans la clause WHERE d'une requête de suppression. En fait, soit vous donnez à votre application le droit de supprimer des enregistrements, soit non. En admettant que vous autorisiez la suppression, je ne vois pas en quoi la modification de la ligne par un autre utilisateur la rende subitement moins supprimable. Il s'agit à mes yeux d'une aberration logique. Car en supposant que le timing soit légèrement différent, j'aurais alors supprimé une ligne qui n'aurait pas du l'être, sans qu'il existe d'indication permettant de discerner une ligne qui peut être supprimée d'une qui ne peut pas l'être. C'est pourquoi dans mes requêtes DELETE vous ne verrez que le champ clé dans le WHERE.

Champ Long String (Memo) ou Long Binary (BLOB)

Je ne vous ai pas parlé de la manipulation de ce type de champ car elle ne présente pas de difficultés majeures.

Contrairement à ADO, il n'y a pas besoin d'utiliser les méthodes GetChunck ou AppendChunck. Vous pouvez manipuler ces champs directement.

```
DataRow MaLigne; string strMemo; byte[] TabBLOB;
strMemo = (string)(MaLigne("ChampMemo"));
TabBLOB = (byte[])(MaLigne("ChampBLOB")) ;
//n'essayez pas TabBLOB = (byte)(MaLigne("ChampBLOB"))
```

L'aide en ligne explique parfaitement la récupération et la transmission vers les champs longs.

Notre problème ici consiste à la manipulation de ces champs vers ou depuis la source de données. Ces champs longs peuvent être d'une taille colossale. Essayez le rapatriement de 100000 enregistrements contenant chacun un champ BLOB de 300 Mo et vous allez voir arriver votre administrateur avec un merlin.

C'est pour contourner le problème que l'on pratique la récupération différée. Il existe deux possibilités :

- On casse la table en deux dans le Dataset, une table contient la clé et tous les champs court, l'autre la clé et tous les champs longs. Une relation un-à-un unit ces deux tables.
- On ne rapatrie que les champs courts. Puis on change la structure de la table dans le Dataset pour ajouter les champs longs. On travaille alors avec deux DataAdapter pour une même table, l'un gérant les champs courts et l'autre les longs.

Quelle que soit la méthode choisie, nous avons pour l'instant juste élaboré la base de notre stratégie. La difficulté réside dans la nécessité de récupérer le champ BLOB uniquement lorsque l'utilisateur en a vraiment besoin. Il faut alors faire preuve d'imagination dans votre interface. Un exemple standard est l'affichage d'un enregistrement dans la feuille avec un bouton 'Afficher le plan' qui lui provoque la récupération de l'image dans cet exemple.

Il suffit alors de rapatrier le champ de l'enregistrement en cours, et tout le monde est content.

L'avantage de cette méthode réside aussi dans le fait que la plupart des champs longs sont rarement modifiés. Par un emploi judicieux de GetChanges, on pourra diminuer ainsi fortement le volume transmis.

Maintenant il est tout à fait possible de laisser la table telle qu'elle est, et de rapatrier les champs longs.

Toutefois il convient de s'assurer que les champs longs ne sont pas inclus dans le critère des commandes de mises à jour sous peine d'une perte de performance totalement inutile.

Ecriture des commandes avec des requêtes paramétrées

Je vais ici donner des exemples de requêtes de DataAdapter. Accrochez-vous car à partir d'ici, il vaut mieux avoir intégré correctement ce que nous avons déjà vu.

Pour commencer, nous allons travailler sur la table Titles dont le schéma est le suivant :

Titles

Nom champ	*Type de données*	*Longueur*	*Précision*	*Commentaires*
Title	AdWChar	255	0	Not NULL
Year Published	AdSmallInt	0	5	
ISBN	adWChar	20	0	Clé primaire
PubId	AdInteger	0	10	Clé étrangère
Description	AdWChar	20	0	
Notes	AdWChar	50	0	
Subject	AdWChar	50	0	
Comments	AdLongVarWChar	0	0	Mémo

La première chose est de créer notre Dataset.

```
this.Form1.Load += new System.EventHandler(Form1_Load);
…
private DataSet dtsBiblio = new DataSet();

private void Form1_Load(object sender, System.EventArgs e) {
   OleDbConnection MaConn = new
OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;");
   OleDbCommand MaCommand = new OleDbCommand("SELECT ISBN, Title, [Year
Published], Subject, Description, Notes, PubID FROM Titles", MaConn);
   dtAdTitre.SelectCommand = MaCommand;
   dtAdTitre.MissingSchemaAction = MissingSchemaAction.AddWithKey;
   dtAdTitre.Fill(dtsBiblio, "Titres");
   tblTitres = dtsBiblio.Tables("Titres");
   OleDbDataAdapter dtAdComment = new OleDbDataAdapter("SELECT ISBN FROM
Titles", MaConn);
   dtAdComment.MissingSchemaAction = MissingSchemaAction.AddWithKey;
   tblTitres.Columns("Title").AllowDBNull = false;
   dtAdComment.Fill(dtsBiblio, "Comments") ;
   dtsBiblio.Tables("Comments").Columns.Add("Comments",
System.Type.GetType("System.String")) ;
   dtsBiblio.Relations.Add(new DataRelation("unAun",
dtsBiblio.Tables("Titres").Columns("ISBN"),
dtsBiblio.Tables("Comments").Columns("ISBN"), true));
   dtsBiblio.Relations("unAun").ChildKeyConstraint.AcceptRejectRule =
AcceptRejectRule.Cascade;
   dtsBiblio.Relations("unAun").ChildKeyConstraint.UpdateRule =
Rule.None;
   dtsBiblio.Relations("unAun").ChildKeyConstraint.DeleteRule =
Rule.Cascade;
```

Jusque là rien de bien complexe. Je crée une table 'Titres' qui regroupe tous les champs courts.

Je crée une seconde table contenant juste le champ clé primaire, et j'ajoute un champ 'Comment' de type Mémo ne contenant rien.

Enfin je mets mes deux tables en relation.

Maintenant je vais devoir créer ma logique de mise à jour.

Suppression

Commençons par la plus simple pour nous chauffer.

Comme je vous l'ai déjà dit, pour ma part je n'intègre que les champs clé dans la clause WHERE. Ma requête sera alors :

```
DELETE Titles WHERE ISBN=?
```

J'utilise pour l'instant des espaces réservés. Si je travaillais avec SQL-Server, je serais obligé d'utiliser des paramètres nommés. Ce n'en serait d'ailleurs que plus simple.

Je crée la fonction suivante

```
private OleDbCommand CreateDeleteCommand(ByVal LaConnexion As
OleDbConnection) {
        string strSQL = "DELETE Titles WHERE ISBN=?";
        OleDbCommand cmdDel = new OleDbCommand(strSQL, LaConnexion);
        OleDbParameter MonParam = new OleDbParameter();
        MonParam.ParameterName = "OrISBN";
        MonParam.OleDbType = OleDbType.VarWChar;
        MonParam.Size = 20;
        MonParam.Direction = ParameterDirection.Input;
        MonParam.IsNullable = false;
        MonParam.SourceColumn = "ISBN";
        MonParam.SourceVersion = DataRowVersion.Original;
        cmdDel.Parameters.Add(MonParam);
        return cmdDel;
    }
```

Je détaille ici bien les propriétés de mon paramètre. Profitez-en car lorsqu'il y a beaucoup de paramètres à définir comme nous allons le voir plus loin, je tends à utiliser la notation suivante :

```
cmdDel.Parameters.Add(new OleDbParameter("OrISBN", OleDbType.VarWChar,
20, ParameterDirection.Input, false, (byte)0, (byte)0, "ISBN",
DataRowVersion.Original, null));
```

Dans le cas des espaces réservés, je définis le nom du paramètre par pure bonté d'âme. Je définis par contre correctement le type et la taille, puisqu'il s'agit d'une chaîne, la taille est obligatoire. Je définis surtout le nom de la colonne source et la version. C'est là que se joue tout le travail du DataAdapter. Lors de l'appel de la méthode Update, chaque fois qu'il trouvera une ligne dont l'état est 'Deleted', il récupérera la valeur originale de la colonne ISBN pour valoriser le paramètre, puis il exécutera la requête définie par strSQL.

Bref, du tout automatique.

Notez que dans une requête de suppression, on passe toujours la version originale car il n'existe pas de valeurs courantes

Insertion

Dans mon exemple, celle-ci va être à peine plus complexe, mais croyez bien qu'avant la fin de ce cours, vous la verrez d'un tout autre œil.

Ma requête va être de la forme :

INSERT INTO Titles(ISBN, Title, [Year Published], Subject, Description, Notes, PubID) VALUES (?, ?, ?, ?, ?, ?, ?, ?)

Je vous fais un code avec les deux notations :

```
private OleDbCommand CreateInsertCommand(OleDbConnection LaConnexion) {
        string strSQL = "INSERT INTO Titles(ISBN, Title, [Year
Published], Subject, Description, Notes, PubID) VALUES (?, ?, ?, ?, ?, ?,
?)";
        OleDbCommand cmdIns = new OleDbCommand(strSQL, LaConnexion);
        OleDbParameter ParISBN = new OleDbParameter();
        ParISBN.ParameterName = "CurISBN";
        ParISBN.OleDbType = OleDbType.VarWChar;
        ParISBN.Size = 20;
        ParISBN.Direction = ParameterDirection.Input;
        ParISBN.IsNullable = false;
        ParISBN.SourceColumn = "ISBN";
        ParISBN.SourceVersion = DataRowVersion.Current;
        cmdIns.Parameters.Add(ParISBN);
        OleDbParameter parTitles = new OleDbParameter();
        parTitles.ParameterName = "CurTitle";
        parTitles.OleDbType = OleDbType.VarWChar;
        parTitles.Size = 255;
        parTitles.Direction = ParameterDirection.Input;
        parTitles.IsNullable = false;
        parTitles.SourceColumn = "Titles";
        parTitles.SourceVersion = DataRowVersion.Current;
        cmdIns.Parameters.Add(parTitles)
        OleDbParameter parYear = new OleDbParameter();
        parYear.ParameterName = "CurYear";
        parYear.OleDbType = OleDbType.UnsignedSmallInt;
        parYear.Precision = 5;
        parYear.Direction = ParameterDirection.Input;
        parYear.IsNullable = true;
        parYear.SourceColumn = "year published";
        parYear.SourceVersion = DataRowVersion.Current;
        cmdIns.Parameters.Add(parYear);
        cmdIns.Parameters.Add(new OleDbParameter("curSubject",
OleDbType.VarWChar, 50, ParameterDirection.Input, true, (byte)0, (byte)0,
"Subject", DataRowVersion.Current, null));
        cmdIns.Parameters.Add(new OleDbParameter("curDescription",
OleDbType.VarWChar, 20, ParameterDirection.Input, true, (byte)0, (byte)0,
"Description", DataRowVersion.Current, null));
        cmdIns.Parameters.Add(new OleDbParameter("curNotes",
OleDbType.VarWChar, 50, ParameterDirection.Input, true, (byte)0, (byte)0,
"Notes", DataRowVersion.Current, null));
        cmdIns.Parameters.Add(new OleDbParameter("PubId",
OleDbType.Integer, 0, ParameterDirection.Input, false, 10, (byte)0,
"PubId", DataRowVersion.Current, null));
        return cmdIns;
    }
```

Dans ce cas, j'utilise les valeurs courantes car les lignes ajoutées n'ont par de valeur originale.

Bien sur ce code est un peu fastidieux à écrire, mais vous pouvez utiliser le code généré par l'assistant et le modifier. J'ai détaillé le début du code pour indiquer deux trois précisions.

- Lorsqu'un champ utilise un caractère réservé comme l'espace, vous ne devez pas le mettre entre crochet dans la propriété SourceColumn.
- Vous n'êtes pas obligé de définir la précision pour les types numériques qui ne sont pas du type `OleDbType.Decimal`.
- Attention de bien mettre la propriété IsNullable du paramètre PubId à false, puisqu'il s'agit d'une clé étrangère pour une autre table.
- Vous n'êtes pas tenu de déclarer la version, puisque la version courante est la valeur par défaut

Modification

C'est hélas la plus fastidieuse à écrire (j'en vois qui regrettent déjà le moteur de curseur).
Il n'y a qu'un piège avec elle, et comme il est évident, il suffit juste d'y penser.
La requête va être de la forme

UPDATE Titles SET ISBN = ?, Title = ?, [Year Published] = ?, Subject = ?, Description = ?, Notes = ?, PubID = ? WHERE ISBN = ? AND Description = ? AND Notes = ? AND PubID = ? AND Subject = ? AND Title = ? AND [Year Published] = ?

Oui je sais, cela fait mal aux yeux. Mais dites-vous bien, que le pire est que cette requête est fausse. En effet, certaines valeurs des champs d'origine peuvent être NULL. Or si vous écrivez une égalité de la forme Champ=NULL cela sera toujours faux. Ce qui revient à dire que toute ligne ayant une valeur d'origine NULL ne pourrait être identifier par cette requête. Le code correcte est donc :

UPDATE Titles SET ISBN = ?, Title = ?, [Year Published] = ?, Subject = ?, Description = ?, Notes = ?, PubID = ? WHERE (ISBN = ?) AND (Description = ? OR ? IS NULL AND Description IS NULL) AND (Notes = ? OR ? IS NULL AND Notes IS NULL) AND (PubID = ?) AND (Subject = ? OR ? IS NULL AND Subject IS NULL) AND (Title = ?) AND ([Year Published] = ? OR ? IS NULL AND [Year Published] IS NULL)

A chaque fois qu'un champ peut être NULL le critère doit être écrit sous la forme :
(Champ = ? OR ? IS NULL AND Champ IS NULL)
Que l'on peut lire sous la forme, le champ est égal au paramètre ou le champ et le paramètre sont NULL.
Il faut bien sur passer tous les paramètres. Dans cette requête qui concerne neuf champs il y a dix-neuf paramètres à transmettre. Cela peut sembler abominable, mais vous allez bientôt voir que le jeu en vaut la chandelle.
J'aurais alors ma fonction.

```
 private OleDbCommand CreateUpdateCommand(OleDbConnection LaConnexion) {
        string strSQL = "UPDATE Titles SET ISBN = ?, Title = ?, [Year
Published] = ?, Subject = ?, Description = ?, Notes = ?, PubID = ? WHERE
(ISBN = ?) AND (Description = ? OR ? IS NULL AND Description IS NULL) AND
(Notes = ? OR ? IS NULL AND Notes IS NULL) AND (PubID = ?) AND (Subject =
? OR ? IS NULL AND Subject IS NULL) AND (Title = ?) AND  ([Year
Published] = ? OR ? IS NULL AND [Year Published] IS NULL)";
        OleDbCommand cmdUpd = new OleDbCommand(strSQL, LaConnexion);
        cmdUpd.Parameters.Add(New OleDbParameter("ISBN",
OleDbType.VarWChar, 20, "ISBN"));
        cmdUpd.Parameters.Add(new OleDbParameter("Title",
OleDbType.VarWChar, 255, "Title"));
        cmdUpd.Parameters.Add(new OleDbParameter("Year_Published",
OleDbType.SmallInt, 0, "Year Published"));
        cmdUpd.Parameters.Add(new OleDbParameter("Subject",
OleDbType.VarWChar, 50, "Subject"));
        cmdUpd.Parameters.Add(new OleDbParameter("Description",
OleDbType.VarWChar, 50, "Description"));
        cmdUpd.Parameters.Add(new OleDbParameter("Notes",
OleDbType.VarWChar, 50, "Notes"));
```

```
cmdUpd.Parameters.Add(new OleDbParameter("PubID", OleDbType.Integer, 0,
"PubID"));
        cmdUpd.Parameters.Add(new OleDbParameter("Original_ISBN",
OleDbType.VarWChar, 20, ParameterDirection.Input, false, (byte)0,
(byte)0, "ISBN", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new OleDbParameter("Original_Description",
OleDbType.VarWChar, 50, ParameterDirection.Input, false, (byte)0,
(byte)0, "Description", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new OleDbParameter("Original_Description1",
OleDbType.VarWChar, 50, ParameterDirection.Input, false, (byte)0,
(byte)0, "Description", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new OleDbParameter("Original_Notes",
OleDbType.VarWChar, 50, ParameterDirection.Input, false, (byte)0,
(byte)0, "Notes", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new OleDbParameter("Original_Notes1",
OleDbType.VarWChar, 50, ParameterDirection.Input, false, (byte)0,
(byte)0, "Notes", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new OleDbParameter("Original_PubID",
OleDbType.Integer, 0, ParameterDirection.Input, false, (byte)0, (byte)0,
"PubID", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new OleDbParameter("Original_Subject",
OleDbType.VarWChar, 50, ParameterDirection.Input, false, (byte)0,
(byte)0, "Subject", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new OleDbParameter("Original_Subject1",
OleDbType.VarWChar, 50, ParameterDirection.Input, false, (byte)0,
(byte)0, "Subject", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new OleDbParameter("Original_Title",
OleDbType.VarWChar, 255, ParameterDirection.Input, false, (byte)0,
(byte)0, "Title", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new
OleDbParameter("Original_Year_Published", OleDbType.SmallInt, 0,
ParameterDirection.Input, false, (byte)0, (byte)0, "Year Published",
DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new
OleDbParameter("Original_Year_Published1", OleDbType.SmallInt, 0,
ParameterDirection.Input, false, (byte)0, (byte)0, "Year Published",
DataRowVersion.Original, null));
        return cmdUpd;
    }
```

Bien sur, je n'ai pas écrit ce code. Je l'ai généré à l'aide de l'assistant et légèrement adapté, cela ne m'a pas pris plus de deux minutes. Maintenant je n'ai plus qu'à ajouter après mon code de création

```
        dtAdTitre.DeleteCommand = CreateDeleteCommand(MaConn);
        dtAdTitre.InsertCommand = CreateInsertCommand(MaConn);
        dtAdTitre.UpdateCommand = CreateUpdateCommand(MaConn);
```

Et mon DataAdapter possède sa logique de mise à jour. Ce n'est pas bien compliqué, mais ce n'est que le début.

Ecriture des commandes avec des procédures stockées

Nous allons reprendre le même exemple ici en utilisant SQL-Server et des procédures stockées. La table 'Titles' de la base de données 'Pubs' pour SQL-Server est différente de celle de la base 'biblio' d'Access mais cela n'a que peut d'importance. De manière générale et sur de nombreux aspects, il est beaucoup plus simple de travailler avec des procédures stockées. Cela diminue grandement le travail de gestion à effectuer avec le code. Donc c'est reparti pour un tour.

```
this.Form1.Load += new System.EventHandler(Form1_Load);
    private DataSet dtsBiblio = new DataSet();
    private void Form1_Load(object sender, System.EventArgs e) {
        SqlConnection MaConn = new SqlConnection(@"packet size=4096;user
id=sa;data source=NOM-SZ71L7KTH17\\TUTO;persist security
info=False;initial catalog=pubs;password=monpasse");
        SqlCommand MaCommand = new SqlCommand();
        MaCommand.CommandText = "psSelectTitre";
        MaCommand.CommandType = CommandType.StoredProcedure;
        MaCommand.Connection = MaConn;
        dtAdTitre.SelectCommand = MaCommand;
        dtAdTitre.MissingSchemaAction = MissingSchemaAction.AddWithKey;
        dtAdTitre.Fill(dtsBiblio, "Titres") ;
        tblTitres = dtsBiblio.Tables("Titres") ;
```

Où la procédure stockée psSelectTitre est définie telle que:

```
CREATE PROCEDURE dbo.psSelectTitre
AS
        SET NOCOUNT ON;
SELECT title_id, title, type, pub_id, price, royalty, ytd_sales, notes, pubdate FROM titles
GO
```

Suppression

La procédure appelée est décrite comme :

```
CREATE PROCEDURE dbo.psDeleteTitre
(       @Original_title_id nvarchar(50)  )
AS
        SET NOCOUNT OFF;
DELETE FROM titles WHERE title_id = @Original_title_id
GO
```

Avec la fonction de création

```
private SqlCommand CreateDeleteCommand(SqlConnection LaConnexion) {
     SqlCommand cmdDel = new SqlCommand("psDeleteTitre", LaConnexion);
     cmdDel.CommandType = CommandType.StoredProcedure;
     SqlParameter MonParam = new SqlParameter();
     MonParam.ParameterName = "Original_title_id";
     MonParam.SqlDbType = SqlDbType.NVarChar;
     MonParam.Size = 50;
     MonParam.Direction = ParameterDirection.Input;
     MonParam.IsNullable = false;
     MonParam.SourceColumn = "title_id";
     MonParam.SourceVersion = DataRowVersion.Original;
     cmdDel.Parameters.Add(MonParam);
     return cmdDel}
```

Jusque là la similitude est frappante. Mais les différences vont rapidement arriver. N'oubliez pas tout de même de bien préciser le type de la commande.

Insertion

Je vais déjà anticiper la suite, donc cette procédure est déjà plus complexe que le strict nécessaire. Nous y reviendrons plus loin, mais notez déjà la présence indispensable de la mise à Off de NOCOUNT afin que le DataAdapter puisse récupérer correctement le nombre de lignes modifiées. Il me suffirait de le remettre à ON juste après pour pouvoir insérer sans risques des enregistrements dans une table de suivie par exemple.

```
CREATE PROCEDURE dbo.psInsertTitre
(
	@title_id nvarchar(50),
	@title nvarchar(80),
	@type nchar(12),
	@pub_id nchar(4),
	@price money,
	@royalty int,
	@ytd_sales int,
	@notes nvarchar(200),
	@pubdate datetime
)
AS
	SET NOCOUNT OFF;
INSERT INTO titles(title_id, title, type, pub_id, price, royalty, ytd_sales, notes, pubdate) VALUES
(@title_id, @title, @type, @pub_id, @price, @royalty, @ytd_sales, @notes, @pubdate);
	SELECT title_id, title, type, pub_id, price, royalty, ytd_sales, notes, pubdate FROM titles WHERE
(title_id = @title_id)
GO
```

Le code de création sera alors :

```
private SqlCommand CreateInsertCommand(SqlConnection LaConnexion) {
        SqlCommand cmdIns = new SqlCommand("psInsertTitre", LaConnexion);
        cmdIns.CommandType = CommandType.StoredProcedure ;
        cmdIns.Parameters.Add(new SqlParameter("@title_id",
SqlDbType.NVarChar, 50, "title_id")) ;
        cmdIns.Parameters.Add(new SqlParameter("@title",
SqlDbType.NVarChar, 80, "title"));
        cmdIns.Parameters.Add(new SqlParameter("@type",
SqlDbType.NVarChar, 12, "type"));
        cmdIns.Parameters.Add(new SqlParameter("@pub_id",
SqlDbType.NVarChar, 4, "pub_id"));
        cmdIns.Parameters.Add(new SqlParameter("@price", SqlDbType.Money,
4, "price"));
        cmdIns.Parameters.Add(new SqlParameter("@royalty", SqlDbType.Int,
4, "royalty"));
        cmdIns.Parameters.Add(new SqlParameter("@ytd_sales",
SqlDbType.Int, 4, "ytd_sales"));
        cmdIns.Parameters.Add(new SqlParameter("@notes",
SqlDbType.NVarChar, 200, "notes"));
        cmdIns.Parameters.Add(new SqlParameter("@pubdate",
SqlDbType.DateTime, 8, "pubdate"));
        return cmdIns;
    }
```

Notez que dans ce cas j'utilise des paramètres nommés. L'ordre d'ajout à la collection n'a plus aucune importance.

Modification

```
CREATE PROCEDURE dbo.psUpdateTitre
(
	@title_id nvarchar(50),
	@title nvarchar(80),
	@type nchar(12),
	@pub_id nchar(4),
	@price money,
	@royalty int,
	@ytd_sales int,
	@notes nvarchar(200),
	@pubdate datetime,
	@Original_title_id nvarchar(50),
	@Original_notes nvarchar(200),
	@Original_price money,
	@Original_pub_id nchar(4),
	@Original_pubdate datetime,
	@Original_royalty int,
	@Original_title nvarchar(80),
	@Original_type nchar(12),
	@Original_ytd_sales int
)
AS
	SET NOCOUNT OFF;
UPDATE titles SET title_id = @title_id, title = @title, type = @type, pub_id = @pub_id, price = @price, royalty = @royalty, ytd_sales = @ytd_sales, notes = @notes, pubdate = @pubdate WHERE (title_id = @Original_title_id) AND (notes = @Original_notes OR @Original_notes IS NULL AND notes IS NULL) AND (price = @Original_price OR @Original_price IS NULL AND price IS NULL) AND (pub_id = @Original_pub_id OR @Original_pub_id IS NULL AND pub_id IS NULL) AND (pubdate = @Original_pubdate) AND (royalty = @Original_royalty OR @Original_royalty IS NULL AND royalty IS NULL) AND (title = @Original_title) AND (type = @Original_type) AND (ytd_sales = @Original_ytd_sales OR @Original_ytd_sales IS NULL AND ytd_sales IS NULL);
	SELECT title_id, title, type, pub_id, price, royalty, ytd_sales, notes, pubdate FROM titles WHERE (title_id = @title_id)
GO
```

Maintenant vous n'avez plus besoin d'explication donc :

```
private SqlCommand CreateUpdateCommand(SqlConnection LaConnexion) {
        SqlCommand cmdUpd = new SqlCommand("psInsertTitre",
LaConnexion) ;
        cmdUpd.CommandType = CommandType.StoredProcedure ;
        cmdUpd.Parameters.Add(new SqlParameter("@RETURN_VALUE",
SqlDbType.Int, 4, ParameterDirection.ReturnValue, false, (byte)0,
(byte)0, "", DataRowVersion.Current, null));
        cmdUpd.Parameters.Add(new SqlParameter("@title_id",
SqlDbType.NVarChar, 6, "title_id"));
        cmdUpd.Parameters.Add(new SqlParameter("@title",
SqlDbType.NVarChar, 80, "title"));
        cmdUpd.Parameters.Add(new SqlParameter("@type",
SqlDbType.NVarChar, 12, "type"));
        cmdUpd.Parameters.Add(new SqlParameter("@pub_id",
SqlDbType.NVarChar, 4, "pub_id"));
        cmdUpd.Parameters.Add(new SqlParameter("@price", SqlDbType.Money,
8, "price"));
```

```
cmdUpd.Parameters.Add(new SqlParameter("@advance", SqlDbType.Money, 8,
"advance"));
        cmdUpd.Parameters.Add(new SqlParameter("@royalty", SqlDbType.Int,
4, "royalty"));
        cmdUpd.Parameters.Add(new SqlParameter("@ytd_sales",
SqlDbType.Int, 4, "ytd_sales"));
        cmdUpd.Parameters.Add(new SqlParameter("@notes",
SqlDbType.NVarChar, 200, "notes"));
        cmdUpd.Parameters.Add(new SqlParameter("@pubdate",
SqlDbType.DateTime, 8, "pubdate"));
        cmdUpd.Parameters.Add(new SqlParameter("@Original_title_id",
SqlDbType.NVarChar, 6, ParameterDirection.Input, false, (byte)0, (byte)0,
"title_id", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new SqlParameter("@Original_advance",
SqlDbType.Money, 8, ParameterDirection.Input, false, (byte)0, (byte)0,
"advance", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new SqlParameter("@Original_notes",
SqlDbType.NVarChar, 200, ParameterDirection.Input, false, (byte)0,
(byte)0, "notes", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new SqlParameter("@Original_price",
SqlDbType.Money, 8, ParameterDirection.Input, false, (byte)0, (byte)0,
"price", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new SqlParameter("@Original_pub_id",
SqlDbType.NVarChar, 4, ParameterDirection.Input, false, (byte)0, (byte)0,
"pub_id", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new SqlParameter("@Original_pubdate",
SqlDbType.DateTime, 8, ParameterDirection.Input, false, (byte)0, (byte)0,
"pubdate", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new SqlParameter("@Original_royalty",
SqlDbType.Int, 4, ParameterDirection.Input, false, (byte)0, (byte)0,
"royalty", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new SqlParameter("@Original_title",
SqlDbType.NVarChar, 80, ParameterDirection.Input, false, (byte)0,
(byte)0, "title", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new SqlParameter("@Original_type",
SqlDbType.NVarChar, 12, ParameterDirection.Input, false, (byte)0,
(byte)0, "type", DataRowVersion.Original, null));
        cmdUpd.Parameters.Add(new SqlParameter("@Original_ytd_sales",
SqlDbType.Int, 4, ParameterDirection.Input, false, (byte)0, (byte)0,
"ytd_sales", DataRowVersion.Original, null));
        return cmdUpd;
}
```

Remarquons tout de même qu'il y a moins besoin de paramètres dans la collection puisque nous travaillons avec des paramètres nommés et plus avec des espaces réservés.

Gestion des modifications

Comme je vous le disais précédemment, lors de l'appel de sa méthode Update, le DataAdapter va parcourir la collection des DataRow de ma table. Dès qu'il trouve une ligne dont le RowState n'est pas 'Unchanged', il appelle la commande correspondante à l'état de la ligne, valorise les paramètres et exécute la commande. La de deux choses l'une. Ou il récupère une valeur positive pour le nombre de lignes modifiées par la commande et celle-ci est considérée comme ayant aboutie ; ou il récupère 0 et il considère celle-ci comme un échec.

La méthode Update

Regardons plus en détail la méthode Update de l'objet DataAdapter. Comme nous l'avons déjà vu, un Dataset n'est jamais lié à une source de données donc jamais à un DataAdapter. L'appel de la méthode Update doit donc toujours faire référence à sa cible. Je vous ai dit plus haut que l'on utilise généralement un DataAdapter par table. Cela est très vrai pour les composants mais la gestion par le code peut être différente. Il s'agit alors de savoir si on préfère créer de nombreux DataAdapter et gérer moins de code ou en créer moins et manipuler plus de commandes.

Les surcharges possibles de la méthode sont :

public int Update(DataRow[] dataRows)

La cible est un tableau d'objet DataRow, souvent obtenu à l'aide de la méthode Select. Un tableau d'objet DataRow ne doit pas être vu comme autre chose qu'un tableau de pointeur visant les lignes de la table dont elles sont issues. Toutes modifications des lignes du tableau sont en fait des modifications des lignes de la table.

public int Update(DataSet dataSet)

Celle-là est un peu plus sournoise. Passer un Dataset comme cible présuppose que l'on a correctement géré le TableMapping du Dataset. En effet, c'est grâce à cette collection que le Dataset saura quelle table fournir.

public int Update(DataTable dataTable)

C'est la surcharge la plus utilisée. Attention, elle attend l'objet DataTable, pas le nom de la table

public int Update(DataRow[] dataRows, DataTableMapping tableMapping)

Plus rarement utilisée, elle attend un tableau d'objets DataRow et un DataTableMapping.

public int Update(DataSet dataSet, string srcTable)

C'est sûrement la plus utilisée. Globalement on travaille principalement avec la première, la troisième et celle-ci.

Donc maintenant j'appelle ma méthode.

Comme je vous l'ai déjà dit, il y a parcours de la Table à la recherche des lignes ayant des modifications en attentes. Si votre table contient un million d'enregistrements et que vous en avez changé trois, ce n'est pas bien rentable. Il existe deux méthodes pour restreindre ce comportement.

Restriction par Select

C'est une méthode très efficace que nous allons revoir bientôt. La méthode permet de mettre un filtre sur l'état / version des lignes. Prenons l'exemple suivant :

```
DataRow MaLign;object[] objTabTemp(6) = new object[6];
for(int compteur = 1 ; compteur <= 2 ; compteur++) {
    objTabTemp(0) = string.Concat(compteur.ToString(),
compteur.ToString(), compteur.ToString(), compteur.ToString());
    objTabTemp(1) = Choose(compteur, "Oui-Oui fait du ski", "inside the
actor's studio") ;
    objTabTemp(2) = Choose(compteur, 1996, 2000) ;
    objTabTemp(3) = DBNull.Value ;
    objTabTemp(4) = "pas de description" ;
    objTabTemp(5) = "mauvaise note" ;
    objTabTemp(6) = 192 ;
    MaLigne = tblTitres.NewRow();
    MaLigne.ItemArray = objTabTemp;
    tblTitres.Rows.Add(MaLigne);
}
int NbModif = dtAdTitre.Update(tblTitres.Select("", "",
DataViewRowState.Added)) ;
```

Dans ce cas mon DataAdapter ne va parcourir que les lignes ajoutées. Cela permet non seulement de réduire considérablement le temps de travail du DataAdapter, mais aussi de pouvoir hiérarchiser les actions (Ajouter avant de supprimer par exemple). N'oubliez pas qu'il est possible de combiner les conditions, par exemple la ligne ci-dessous est valide :

```
int NbModif = dtAdTitre.Update(tblTitres.Select("", "",
DataViewRowState.Added | DataViewRowState.Deleted));
```

Restriction par GetChanges

La méthode GetChanges renvoie une copie du Dataset ne contenant que les lignes modifiées et les lignes nécessaires à l'intégrité référentielle de celles-ci. Attention, le résultat et un Dataset. Contrairement à un tableau de DataRow, il véhicule aussi des contraintes, des relations et peut être synchronisé avec un XMLDataDocument.

Il est aussi possible de restreindre le type de changement en passant un paramètre ou une combinaison à GetChanges

```
int NbModif = dtAdTitre.Update(dtsBiblio.GetChanges(DataRowState.Added));
```

Ces deux méthodes ont pourtant d'autres différences lors de la récupération des informations que nous allons voir maintenant.

Récupération des informations de mise à jour

Update : Une opération globale

Il existe une certaine dualité dans le fonctionnement de la méthode Update. Il s'agit d'une opération globale qui regroupe une somme d'opérations atomiques. Votre premier choix dans l'élaboration de votre stratégie de mise à jour est : "quel comportement adopter lors de l'échec d'une mise à jour ?".

Dans la gestion des exceptions nous verrons un certain nombre de problèmes que vous pouvez rencontrer, mais si vous travaillez correctement, seule l'exception 'DBConcurrencyException' peut intervenir lors d'une opération de mise à jour.

Dans toute cette partie, je ne vous donne que les aspects techniques de la mise à jour. A ma connaissance, il n'y a pas de bonnes ou de mauvaises stratégies de mise à jour tant que l'intégrité et la cohérence de la source de données sont respectées. Le choix final peut dépendre, de votre application, des droits que vous possédez, de votre vision de la problématique mais nullement d'une conduite que je vous aurais suggérée, car selon mes applications, je ne procède pas toujours au même choix.

Interrompre ou poursuivre

Lorsqu'une des commandes du DataAdapter obtient zéro modification lors de l'exécution de la requête, il y a lecture de la propriété **ContinueUpdateOnError.** Celle-ci définit le comportement global que vous avez décidé d'adopter.

- Si la valeur est 'True', il y a remontée d'un message d'erreur vers la propriété RowError de la ligne et poursuite éventuelle de l'opération.
- Si sa valeur est 'False', il y a levée de l'exception 'DBConcurrencyException' (normalement) et l'opération de mise à jour est interrompue. Il existe alors deux possibilités :
  - Poursuivre l'opération en corrigeant éventuellement l'erreur
  - Arrêter l'opération et invalider les modifications
  - Il existe un troisième choix, inacceptable à mes yeux, qui consisterait à laisser les données en l'état et d'interrompre l'opération. Comme dans un contexte de concurrence il n'est pas possible de prévoir où se produiront les erreurs, cela reviendrait à laisser la source dans un état incohérent.

Les deux modes de fonctionnement reposent sur une stratégie assez différente même si le comportement final est similaire. Pour ma part, je préfère travailler en mode continu et avec des évènements de ligne que de lever une exception. Toutefois lorsque je travaille en transaction globale, j'utilise aussi la levée d'exception.

Je vais vous donner un exemple dans le paragraphe suivant.

Transaction globale, transactions partielles

Je ne vais pas détailler ici ce qu'est une transaction ni ce qu'elle implique. Pour schématiser, disons qu'une transaction permet de traiter un groupe d'actions comme une entité atomique (non-divisible) ; c'est à dire que toutes les modifications seront acceptées ou toutes seront rejetées. Ceci permet une approche sécurisée d'un groupe d'actions. Maintenant il existe plusieurs approches utilisant les transactions.

La transaction globale

Ceci consiste à traiter l'opération Update comme une entité atomique. En général on utilise alors la levée d'exception pour annuler la transaction. L'objet DataAdapter n'expose pas de propriétés Transaction, vous devez donc la spécifier pour tout ou partie de ces objets Command.

```
OleDbTransaction MaTransac = MaConn.BeginTransaction() ;
dtAdTitre.DeleteCommand.Transaction = MaTransac ;
dtAdTitre.InsertCommand.Transaction = MaTransac ;
dtAdTitre.UpdateCommand.Transaction = MaTransac ;
try {
    dtAdTitre.Update(tblTitres) ;
    MaTransac.Commit() ;
    tblTitres.AcceptChanges() ;
    MessageBox.Show("Modifications acceptées", "MODIFICATION",
MessageBoxButtons.OK, MessageBoxIcon.Information);
catch(DBConcurrencyException ConcOptEx) {
    MaTransac.Rollback();
    tblTitres.RejectChanges() ;
    MessageBox.Show("Modifications rejetées, les tables ont été remises
dans l'état d'origine", "ERREUR", MessageBoxButtons.OK,
MessageBoxIcon.Error);
}
catch(Exception Ex) {
    //gestion des autres erreurs
}
```

Si vous utilisez le CommandBuilder, vous devez le forcer à générer ses commandes explicitement pour les attribuer à la transaction.

```
OleDbCommandBuilder CmdBuild = new OleDbCommandBuilder(dtAdTitre);
CmdBuild.GetDeleteCommand.Transaction = MaTransac;
CmdBuild.GetInsertCommand.Transaction = MaTransac;
CmdBuild.GetUpdateCommand.Transaction = MaTransac;
```

Par défaut, le CommandBuilder ne génère ses commandes que lors de l'appel Update, d'où la nécessité de cette écriture.

Les transactions partielles

Ceci englobe deux notions différentes, les transactions successives et les transactions imbriquées.

Les transactions successives consistent à gérer des lots Update comme des transactions indépendantes. On peut décider de faire une transaction pour les ajouts puis une pour les modifications.

Les transactions imbriquées induisent une hiérarchie dans la transaction. Elles sont en générale dans le traitement par ligne. On crée une transaction mère qui contient d'autres transactions. Il convient alors d'être particulièrement vigilant au niveau d'isolation utilisé.

Update : Une succession d'opération atomique

A chaque ligne ayant des modifications en attente, la suite d'évènements suivants va se produire :

1. Le DataAdapter examine l'état de la ligne et sélectionne la commande à utiliser. Si celle-ci n'est pas définie, il y a appel du CommandBuilder. Si celui-ci n'est pas défini, il y a levée d'une exception 'InvalidOperationException'.
2. Les paramètres dont la propriété 'SourceColumn' est définie sont valorisés
3. L'événement 'OnRowUpdating' est déclenché. Dans les arguments de l'événement vous pouvez récupérer

Command	Obtient ou définit l'objet Command à exécuter en même temps que Update.
Errors	Obtient les erreurs générées par le fournisseur de données .NET Framework lors de l'exécution de Command.
Row	Obtient le DataRow à envoyer par l'intermédiaire de Update.
StatementType	Obtient le type de l'instruction SQL à exécuter (Insert, Update ou Delete)
Status	Obtient UpdateStatus de la propriété Command.
TableMapping	Obtient le DataTableMapping à envoyer par l'intermédiaire de Update.

4. La commande s'exécute
5. Il y a lecture de la propriété UpdateStatus de la commande utilisée. Selon celle-ci, il y a retour des informations vers la ligne
6. L'événement 'OnRowUpdated' est déclenché. Dans les arguments de l'événement vous pouvez récupérer les mêmes arguments que précédemment plus le nombre de lignes modifiées dans 'RecordsAffected'
7. La méthode AcceptChanges est appelée

Il y a donc possibilité d'exercer un contrôle sur chaque opération de ligne grâce à la gestion des deux évènements RowUpdating et RowUpdated. Cette gestion est une alternative à la transaction globale. Elle est particulièrement puissante et permet de construire des stratégies particulières pour chaque application, mais elle peut être lourde à mettre en place.

Mettre à jour le Dataset

Dans deux cas au moins il y a nécessité de récupérer des informations supplémentaires du SGBD.

- Les valeurs de version (TimeStamp) : Lorsque la mise à jour est réussie, le SGBD crée une nouvelle valeur de version pour la ligne. Si vous ne récupérez pas celle-ci, vous vous mettez dans l'impossibilité de modifier une nouvelle fois la ligne.
- Les clés générées : De même que pour les valeurs de versions, avec en plus le besoin de connaître la valeur pour la communiquer aux enregistrements enfants.

Comprendre les clés générées

Il nous faut déjà comprendre correctement comment fonctionnent les propriétés des colonnes de l'objet DataTable. Lorsque je remplis ma table, je peux préciser au DataAdapter de récupérer des informations sur la clé en fixant la propriété MissingSchemaAction à la valeur AddWithKey. Si je fais cela sur une table possédant une colonne auto-incrémentée comme clé primaire, le DataAdapter va attribuer une valeur transparente pour la base de l'incrément, celle-ci n'étant plus modifiable. Un exemple va vous expliquer plus clairement le problème.

```
OleDbConnection MaConn = new
OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;");
OleDbCommand MaCommand = new OleDbCommand("SELECT Au_Id, Author, [year
born] FROM Authors", MaConn);
DataSet dtsBiblio = New DataSet();
OleDbDataAdapter dtAdAuteur = new OleDbDataAdapter() ;
//remplissage de la table
dtAdAuteur.SelectCommand = MaCommand ;
dtAdAuteur.MissingSchemaAction = MissingSchemaAction.AddWithKey ;
dtAdAuteur.Fill(dtsBiblio, "Auteurs") ;
//paramétrage supplémentaire
DataTable tblAuteurs = dtsBiblio.Tables("Auteurs");
tblAuteurs.Columns(0).AutoIncrementSeed = -1;
tblAuteurs.Columns(0).AutoIncrementStep = -1;
//ajout des lignes
DataRow MaLigne;
for(int cmpt = 1 ; cmpt <= 2 ; cmpt++) {
    MaLigne = tblAuteurs.NewRow() ;
    MaLigne("Author") = Choose(cmpt, "Bidou", "Voltaire");
    MaLigne("year born") = Choose(cmpt, 1967, 1694);
    tblAuteurs.Rows.Add(MaLigne) ;
}
```

Ce code va déclencher une erreur sur la ligne tblAuteurs.Rows.Add(MaLigne)

En effet, au moment de l'appel de NewRow, la colonne Au_Id étant de type auto-incrémentée, il y a attribution d'une valeur étant égale au numéro maximum de la colonne +1. Malheureusement comme on a fixé le pas à –1, le numéro créé est le numéro maximum présent dans la colonne qui existe forcément déjà, donc levée d'une exception due à la contrainte d'unicité. Pour faire bref, si vous utilisez AddWithKey, vous ne devez pas modifier les propriétés 'AutoIncrementSeed' et ' AutoIncrementStep'.

Il y a pourtant nécessité à ne pas laisser le DataAdapter gérer les numéros de clé à sa guise, comme nous le verrons un peu plus loin.

Donc quand nous travaillons sur des colonnes de ce type, nous devons suivre les règles suivantes :

- ✓ Toujours veiller à intégrer explicitement la colonne dans la requête select
- ✓ Ne jamais fixer la valeur de MissingSchemaAction à la valeur AddWithKey
- ✓ Gérer les valeurs de l'incrément
- ✓ Toujours utiliser NewRow pour ajouter des lignes

Toute ces difficultés sont intrinsèques au modèle déconnecté.

Replacons le problème. C'est le SGBD qui génère la valeur de clé des lignes ajoutées dans la base de données. Dans le modèle déconnecté, vous récupérer une table dans un état donné, puis la liaison avec la source est coupée. Vous avez donc d'un coté une colonne auto-incrémentée dans la source et de l'autre une colonne de type integer, de type auto-incrémentée ou non dans votre DataTable. La colonne ayant la contrainte d'unicité vous devez seulement veiller à ce qu'une ligne ajoutée dans votre table n'ait pas un numéro existant. Généralement, on utilise une colonne auto-incrémentée dans la DataTable à laquelle on attribue des valeurs négatives afin de détecter les éventuels conflits qui peuvent survenir avec l'utilisation de la méthode Merge.

Mais cela permet aussi de se couvrir contre une faute de priorité dans la programmation. Imaginons le cas suivant :

```
this.Form1.Load += new System.EventHandler(Form1_Load);
    …
    private void Form1_Load(object sender, System.EventArgs e){
        this.DtsPubs1.EnforceConstraints = false;
        this.OleDbDAPublishers.Fill(this.DtsPubs1.Publishers);
        this.OleDbDATitles.Fill(this.DtsPubs1.Titles);
        this.DtsPubs1.EnforceConstraints = true;
        Dim MaLigEd As dtsPubs.PublishersRow =
this.DtsPubs1.Publishers.AddPublishersRow("bidou", "bidou SARL", null,
"69001", "lyon", "france", null, null, null);
        Dim MaLigTit As dtsPubs.TitlesRow =
this.DtsPubs1.Titles.AddTitlesRow("1111", "My sister is not a boy",
MaLigEd.PubID, null, 2000, null, null, null);
        this.OleDbDATitles.Update(this.DtsPubs1.Titles);
    }
```

Dans mon Dataset, je laisse le DataAdapter attribuer le numéro à ma ligne Editeur. Ce numéro est ensuite attribué comme clé étrangère de ma ligne de titre. En l'état, lors de l'appel de la méthode Update sur ma table 'titres', je doit obtenir une erreur puisque je n'ai pas encore ajouté ma ligne éditeur à la source de données. Pourtant si un autre utilisateur a ajouté un éditeur dans ce laps de temps, la commande va passer et mon livre sera attribué à un autre éditeur. Cela ne pourrait être possible si j'avais généré une clé négative.

Car le deuxième problème vient des données relationnelles. Lorsque j'ajoute des données dans des tables liées, mes valeurs de clé primaires générées par ADO vont être utilisées comme clé étrangère dans la table fille. Ces valeurs générées ne seront sûrement pas celles qui seront attribuées par le SGBD. Je vais donc devoir instaurer une hiérarchie à mes ordres de mise à jour, ce qui est normal, mais aussi utiliser une relation en cascade pour que les bonnes valeurs soient attribuées lors des mises à jour;

Nous reverront en détail toutes ces techniques.

Utiliser des requêtes

Commençons par le plus compliqué. Access ne gère ni les paramètres de sortie, ni les procédures stockées, ni les lots de requêtes. Dans ce cas, nous devons gérer le travail de récupération. Le seul moment ou cela est possible, c'est dans la gestion de l'événement OnRowUpdated. Dans notre exemple, il faut utiliser Access 2000 ou supérieur car Access 97 ne supporte pas les requêtes de récupération d'identité.

Le principe consiste à aller récupérer la valeur de clé générée à l'aide de la requête "SELECT @@IDENTITY". Celle-ci est sans risque avec Access puisqu'elle récupère le dernier numéro attribué sur la connexion, et qu'Access ne risque pas d'être perturbé par un trigger, ceux-ci n'existant pas.

Un exemple serait :

```
this.Form1.Load += new System.EventHandler(Form1_Load);
…
private void Form1_Load(object sender, System.EventArgs e){
OleDbConnection MaConn = new
OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;");
OleDbCommand MaCommand = new OleDbCommand("SELECT Au_Id, Author, [year
born] FROM Authors", MaConn);
DataSet dtsBiblio = new DataSet();
OleDbDataAdapter dtAdAuteur = new OleDbDataAdapter();
//remplissage de la table
dtAdAuteur.SelectCommand = MaCommand ;
dtAdAuteur.Fill(dtsBiblio, "Auteurs") ;
//paramétrage supplémentaire
DataTable tblAuteurs = dtsBiblio.Tables("Auteurs") ;
tblAuteurs.Columns(0).AutoIncrement = true;
tblAuteurs.Columns(0).AutoIncrementSeed = -1;
```

```
tblAuteurs.Columns(0).AutoIncrementStep = -1;
tblAuteurs.Columns(0).Unique = true;
tblAuteurs.Columns(0).AllowDBNull = false;
//création de la commande d'insertion
dtAdAuteur.InsertCommand = new OleDbCommand("INSERT INTO Authors
(Author,[year born]) VALUES (?,?)", MaConn);
dtAdAuteur.InsertCommand.Parameters.Add("Auteur", OleDbType.VarWChar, 50,
"Author");
dtAdAuteur.InsertCommand.Parameters.Add("Annee", OleDbType.SmallInt, 0,
"year born");
dtAdAuteur.InsertCommand.UpdatedRowSource = UpdateRowSource.None;
//ajout des lignes
DataRow MaLigne;
for(int cmpt=0; cmpt<2;cmpt++){
    MaLigne = tblAuteurs.NewRow() ;
    MaLigne("Author") = Choose(cmpt, "Bidou", "Voltaire");
    MaLigne("year born") = Choose(cmpt, 1967, 1694);
    tblAuteurs.Rows.Add(MaLigne);
}
//ajout de l'évènement
dtAdAuteur.RowUpdated += new
OleDbRowUpdatedEventHandler(dtAdAuteur_RowUpdated);
dtAdAuteur.Update(tblAuteurs);
DataView VueAuteurs = new DataView(tblAuteurs);
VueAuteurs.Sort = "Au_Id DESC";
this.DataGrid1.DataSource = VueAuteurs;
}

private static void dtAdAuteur_RowUpdated(object sender,
OleDbRowUpdatedEventArgs args){
if(args.RecordsAffected == 1 && args.StatementType ==
StatementType.Insert){
    OleDbCommand ComRecup = new OleDbCommand("SELECT @@IDENTITY",
args.Command.Connection);
    args.Row("Au_Id") = (int)(ComRecup.ExecuteScalar());
    args.Row.AcceptChanges();
}
}
```

Notez bien que dans la ligne

OleDbCommand ComRecup = new OleDbCommand("SELECT @@IDENTITY", args.Command.Connection)

J'utilise bien la connexion de la commande utilisée. Ceci est **indispensable** au bon fonctionnement de la requête IDENTITY.

Pensez à mettre la propriété UpdatedRowSource de la commande à None puisque dans ce cas elle n'est pas utilisée.

Utiliser des Procédures stockées

Dans ce cas, c'est nettement plus simple car globalement il n'y a rien d'autre à faire que d'écrire correctement la procédure. Si vous reprenez les procédures que nous avons utilisées plus haut, vous verrez que toutes finissent par une requête SELECT récupérant la ligne. On pourrait aussi récupérer que certains champs avec des paramètres de sortie. Grâce à cela, nul besoin de gestion évènementielle, il suffit de configurer correctement la propriété UpdatedRowSource. Ainsi pour reprendre mon exemple, il faudrait écrire :

```
dtAdTitre.DeleteCommand = CreateDeleteCommand(MaConn);
dtAdTitre.DeleteCommand.UpdatedRowSource = UpdateRowSource.None;
dtAdTitre.InsertCommand = CreateInsertCommand(MaConn);
dtAdTitre.InsertCommand.UpdatedRowSource =
UpdateRowSource.FirstReturnedRecord;
dtAdTitre.UpdateCommand = CreateUpdateCommand(MaConn);
dtAdTitre.UpdateCommand.UpdatedRowSource =
UpdateRowSource.FirstReturnedRecord;
```

Notez qu'au niveau des performances, il est toujours un peu meilleur d'utiliser des paramètres de sortie plutôt qu'un lot de requêtes.

Gérer les échecs

Malheureusement, il n'y a pas que les succès qui renvoient des informations. Dans l'ensemble la gestion des échecs suit trois cheminements possibles, si j'omets l'annulation par transaction

Informer et rendre la main à l'utilisateur

C'est probablement le cas le plus fréquent. Comme seul l'utilisateur connaît la signification des modifications, on transmet un rapport d'erreur qui peut prendre la forme de votre choix. Voici le code de l'énumération détaillée des erreurs

```
this.cmdMAJ.Click += new System.EventHandler(cmdMAJ_Click);
   private sub cmdMAJ_Click(object sender, System.EventArgs e){
       this.OleDbDATitles.ContinueUpdateOnError = true;
       this.OleDbDATitles.Update(this.DtsPubs1.Titles);
       this.OleDbDAPublishers.Update(this.DtsPubs1.Publishers);
       if(this.DtsPubs1.HasErrors == true){
           DataRow[] RowsErr();
           foreach(DataTable TableTemp in this.DtsPubs1.Tables){
               if(TableTemp.HasErrors == true){
                   RowsErr = TableTemp.GetErrors();
                   for(int cmpt=0;cmpt<RowsErr.GetUpperBound(0);cmpt++){
                       MessageBox.Show("Erreur sur la ligne : " +
RowsErr(cmpt).Item(TableTemp.PrimaryKey(0)) + " : " +
RowsErr(cmpt).RowError);
                    if(RowsErr(cmpt).GetColumnsInError.GetLength(0) > 0){
                        foreach(DataColumn ColErr in TableTemp.Columns) {
                                MessageBox.Show(ColErr.ColumnName + "A
rencontré l'erreur : " + RowsErr(cmpt).GetColumnError(ColErr));
                            }
                        }
                        RowsErr(cmpt).ClearErrors();
                   }
               }
           }
       }
    }
```

Dans ce cas j'efface l'erreur, mais ceci n'a rien d'obligatoire.

Correction dynamique

Autrement plus complexe est la gestion du traitement dynamique. En fait, elle ne peut concerner que de rares cas ou la correction peut être devinée. N'oubliez pas que nous parlons d'erreur de concurrence optimiste c'est à dire de gérer le comportement de votre application en fonction d'actions d'autres utilisateurs par nature imprédictibles. Dans certain cas de modifications relatives de la valeur d'un champ, il est possible d'anticiper sur l'erreur pour légèrement fausser le comportement concurrentiel et s'assurer que la modification va être acceptée. Pour cela on utilise l'événement 'OnRowUpdating' qui se produit avant l'envoi de la commande.

Je ne vais pas vous donner ce genre d'exemple car il s'agit de cas très particuliers, demandant un codage particulièrement rigoureux.

Rejet partiel

Le rejet partiel est un mode très simple et très fonctionnel de comportement prédictif. Il revient à dire que lorsqu'une ligne particulière est rejetée, toutes les lignes étant des modifications connexes doivent être rejetées aussi. On l'utilise principalement dans les mises à jour relationnelles que nous allons voir maintenant.

Mise à jour relationnelle

Les mises à jours relationnelles suivent des règles strictes, aussi ne sont-elles pas difficiles à gérer, pour peu que l'on ait correctement compris le principe.

Pour ne pas violer l'intégrité référentielle, on ne peut pas soumettre les modifications de tables parent-enfant n'importe comment. La soumission d'une table complète même n'est pas viable puisque l'ordre de soumission dépend de la nature de la commande. Je vais dans cette partie reprendre le Dataset fortement typé que nous avons construit auparavant.

Supposons que je veuille ajouter, modifier et supprimer des éditeurs ainsi que des titres, la table titre étant fille de la table éditeurs. Si je fais un Update de la table titre complète, j'aurais des erreurs car je ne pourrais pas ajouter de titre ayant pour éditeur un enregistrement non encore envoyé vers la source. De même, si je fais mon Update sur la table Editeurs, j'aurais des erreurs car je ne peux pas supprimer un éditeurs ayant des livres. Je dois donc toujours suivre le scénario suivant.

I. Envoyer les enregistrements ajoutés à la table mère
II. Envoyer les enregistrements ajoutés à la table fille
III. Envoyer les enregistrements modifiés de la table mère
IV. Envoyer les enregistrements modifiés de la table fille
V. Envoyer les enregistrements supprimés de la table fille
VI. Envoyer les enregistrements supprimés de la table mère

Et ne jamais déroger à cet ordre sous peine de dysfonctionnements majeurs. Bien sur, il y a la possibilité d'utiliser des relations en cascade mais très souvent celles-ci dépassent nos espérances et je vous conseille de les éviter.

Pour illustrer mon propos, je vais enlever l'éditeur n°192 de mon dataset fortement typé.

Donc j'utilise le code suivant :

```
this.DtsPubs1.EnforceConstraints = false;
this.OleDbDAAuthors.Fill(this.DtsPubs1.Authors);
this.OleDbDAPublishers.Fill(this.DtsPubs1.Publishers);
this.OleDbDATitAut.Fill(this.DtsPubs1.Title_Author);
this.OleDbDATitles.Fill(this.DtsPubs1.Titles);
this.DtsPubs1.EnforceConstraints = true;
dtsPubs.PublishersRow MaLigEd =
this.DtsPubs1.Publishers.FindByPubID(192);
DataRow MesLigTitres[] =
MaLigEd.GetChildRows(MaLigEd.Table.ChildRelations(0));
for(int cmpt=0; cmpt<MesLigTitres.GetUpperBound(0);cmpt++){
    MesLigTitres(cmpt).Delete();
}
```

Dès à présent, j'aurais une erreur sur l'appel de la méthode Delete. Ceci sans avoir tenté de mise à jour. Cette erreur est normale puisque mes titres sont engagés dans une relation plusieurs à plusieurs avec les auteurs. Cette erreur est aussi le signe que mon Dataset fortement typé est bien construit. En effet, si j'avais créé des relations en cascade, je n'aurais pas eu l'erreur maintenant, mais lors de la tentative de mise à jour. C'est pourquoi il faut toujours avoir un Schéma au moins aussi restrictif que la source.

Notez au passage la plus grande simplicité du code utilisant un Dataset fortement typé.

Il nous faut donc supprimer les titres de la relation titres-auteurs. Mais pour travailler proprement, nous devons retirer les auteurs qui n'auront plus de titre après la suppression. Si cet exercice n'est pas difficile à réaliser c'est parce que nous connaissons le schéma à la création; mais s'il nous fallait traiter tout cela à l'exécution de façon générique, ce serait une autre paire de manche.

Je vais en profiter pour ajouter aussi une ligne à chaque table et les mettre toutes en relations.

```
this.Form1.Load += new System.EventHandler(Form1_Load);
…
private void Form1_Load(object sender, System.EventArgs e){
this.DtsPubs1.EnforceConstraints = false;
this.OleDbDAAuthors.Fill(this.DtsPubs1.Authors);
this.OleDbDAPublishers.Fill(this.DtsPubs1.Publishers);
this.OleDbDATitAut.Fill(this.DtsPubs1.Title_Author);
this.OleDbDATitles.Fill(this.DtsPubs1.Titles);
this.DtsPubs1.EnforceConstraints = true;
'Exemple de suppression
dtsPubs.PublishersRow MaLigEd =
this.DtsPubs1.Publishers.FindByPubID(192);
dtsPubs.TitlesRow MesLigTitres() =
MaLigEd.GetChildRows(MaLigEd.Table.ChildRelations(0));
int cmpt, cmpt1;
dtsPubs.Title_AuthorRow[] MesLigRelat;
for(cmpt=0;cmpt<MesLigTitres.GetUpperBound(0);cmpt++){
    MesLigRelat =
MesLigTitres(cmpt).GetChildRows("TitlesTitle_x0020_Author");
    For(cmpt1=0;cmpt<MesLigRelat.GetUpperBound(0);cmpt1++){
      MesLigRelat(cmpt1).Delete();
    }
    MesLigTitres(cmpt).Delete();
}
MaLigEd.Delete();
dtsPubs.AuthorsRow mesLigAuteurs() =
this.DtsPubs1.Authors.Select("count(child(AuthorsTitle_x0020_Author).Au_I
d)=0");
for(cmpt=0;cmpt<mesLigAuteurs.GetUpperBound(0);cmpt++){
    mesLigAuteurs(cmpt).Delete() ;
}
'exemple d'ajout
dtsPubs.AuthorsRow MaLigAut = this.DtsPubs1.Authors.NewAuthorsRow;
dtsPubs.TitlesRow MaLigTit = this.DtsPubs1.Titles.NewTitlesRow;
MaLigEd = this.DtsPubs1.Publishers.NewPublishersRow;
MaLigEd.Address = "2 rue du Castor";
MaLigEd.City = "Paris";
MaLigEd.Company_Name = "Edition du Castor";
MaLigEd.Name = "ECSARL";
MaLigEd.SetCommentsNull();
MaLigEd.SetFaxNull();
MaLigEd.SetTelephoneNull();
MaLigEd.State = "FRANCE";
MaLigEd.Zip = "75012";
```

```
this.DtsPubs1.Publishers.AddPublishersRow(MaLigEd);
MaLigAut.Author = "Bidou";
MaLigAut.Year_Born = 1967;
this.DtsPubs1.Authors.AddAuthorsRow(MaLigAut);
MaLigTit.Description = "c'est un bon livre" ;
MaLigTit.ISBN = "1111-2222" ;
MaLigTit.Notes = "si si!" ;
MaLigTit.PubID = MaLigEd.PubID;
MaLigTit.SetCommentsNull();
MaLigTit.Subject = "livre" ;
MaLigTit.Title = "le vin se sifflera trois fois" ;
MaLigTit.Year_Published = "2000";
this.DtsPubs1.Titles.AddTitlesRow(MaLigTit);
this.DtsPubs1.Title_Author.AddTitle_AuthorRow(MaLigAut, MaLigTit);
}

this.cmdMAJ.Click += new System.EventHandler(cmdMAJ_Click);
…
private void cmdMAJ_Click(object sender, System.EventArgs e){
 //les suppressions
this.OleDbDATitAut.Update(this.DtsPubs1.Title_Author.Select("", "",
DataViewRowState.Deleted));
this.OleDbDAAuthors.Update(Me.DtsPubs1.Authors.Select("", "",
DataViewRowState.Deleted));
this.OleDbDATitles.Update(this.DtsPubs1.Titles.Select("", "",
DataViewRowState.Deleted));
Me.OleDbDAPublishers.Update(Me.DtsPubs1.Publishers.Select("", "",
DataViewRowState.Deleted))
//les ajouts
this.OleDbDAPublishers.Update(this.DtsPubs1.Publishers.Select("", "",
DataViewRowState.Added));
this.OleDbDAAuthors.Update(this.DtsPubs1.Authors.Select("", "",
DataViewRowState.Added));
this.OleDbDATitles.Update(this.DtsPubs1.Titles.Select("", "",
DataViewRowState.Added));
this.OleDbDATitAut.Update(this.DtsPubs1.Title_Author.Select("", "",
DataViewRowState.Added));
}

this.OleDbDAPublishers.RowUpdated += new
OleDbRowUpdatedEventHandler(OleDbDAPublishers_RowUpdated) ;
…
private void OleDbDAPublishers_RowUpdated(object sender,
System.Data.OleDb.OleDbRowUpdatedEventArgs e){
  if (e.StatementType == StatementType.Insert){
    OleDb.OleDbCommand MaComRecup = new OleDb.OleDbCommand("SELECT
@@IDENTITY", e.Command.Connection);
    e.Row.Item("PubId") = MaComRecup.ExecuteScalar();
    e.Row.AcceptChanges();
  }
}
this.OleDbDAAuthors.RowUpdated += new
OleDbRowUpdatedEventHandler(OleDbDAAuthors_RowUpdated) ;
…
```

```
private void OleDbDAAuthors_RowUpdated(object sender,
System.Data.OleDb.OleDbRowUpdatedEventArgs e){
  if (e.StatementType == StatementType.Insert){
    OleDb.OleDbCommand MaComRecup = new OleDb.OleDbCommand("SELECT
@@IDENTITY", e.Command.Connection);
    e.Row.Item("Au_Id") = MaComRecup.ExecuteScalar();
    e.Row.AcceptChanges();
  }
}
```

Ce code fonctionnera maintenant parfaitement pour les suppressions. Par contre vous pouvez rencontrer des problèmes lors des insertions. Pourquoi ?

Je récupère bien les valeurs de clés générées par le SGBD dans l'événement RowUpdated. Mais ces valeurs ne vont pas forcément être mises à jour dans les tables filles. Dans le cas que je prends, la propriété UpdateRule vaut Cascade dans toutes les relations, les enregistrements liés récupèrent donc bien la bonne valeur avant d'être envoyés vers la source. Mais si ces relations valaient None cela ne fonctionnerait pas. J'aurais alors une erreur lors de l'insertion puisque la valeur de clé étrangère fournie n'existerait pas. Cette erreur pourrait être facilement contournée dans l'événement RowUdpated par une gestion des enregistrements enfants.

Mise à jour distante

Tout cela est donc parfait dans une application Client / Serveur. Mais il en va autrement pour les architectures n-tiers. Comme je vous l'ai déjà dit dans ce type d'architecture, le fournisseur managé est généralement dans une couche métier distante, et il y a échange XML entre la couche Métier et la couche présentation (votre application).

Si je reprends mon exemple précédent, vous voyez que je vais communiquer toute une base de données pour modifier six enregistrements. Il est probable que vous aurez rapidement un échange de vue un peu vif avec le Rottweiler de l'administrateur si vous travaillez ainsi. Heureusement, ADO.NET propose une solution pour ceux qui n'aiment pas les chiens.

Comme le but est d'économiser la bande passante, vous pouvez appeler la méthode GetChanges de l'objet Dataset. Celle-ci crée un nouveau Dataset contenant toutes les lignes ayant une modification en attente (dont l'état n'est pas 'Unchanged') plus toutes les lignes nécessaires à la garantie de l'intégrité référentielle.

Vous ne trouverez pas de code dans cette partie puisque celui n'a rien de particulier, c'est le mécanisme qu'il faut bien comprendre. Donc je passe mon petit Dataset à ma couche métier, celle-ci applique les modifications et renvoie le Dataset. Je dispose donc de deux objets Dataset, le Dataset principal, et le Dataset entrant. Pour n'en obtenir plus qu'un je vais utiliser la méthode Merge.

Les règles de la fusion sont les suivantes :

- Les tables du Dataset entrant sont comparées aux tables du Dataset principale, s'il y a identité de nom les tables sont fusionnées, sinon la table est ajoutée.
- Quand deux tables sont fusionnées, il y a recherche d'identité de la clé primaire. Si la colonne porte le même nom, il y a fusion, sinon l'événement MergeFailed est levé.
- Pour les lignes ayant la même valeur de clé primaire dans les deux Dataset, il y a écrasement des valeurs **originales** par les valeurs **originales** du Dataset entrant.
- Les lignes du Dataset entrant n'ayant pas de ligne équivalente sont ajoutées.

Comme dans notre cas, le Dataset entrant est un clone partiel du Dataset principal, il y aura identité complète. Donc il faut veiller à :

- Vérifier les lignes en erreurs du Dataset entrant afin de dupliquer l'erreur sur le Dataset principal et/ou d'appliquer RejectChanges sur ces lignes
- Appeler la méthode AcceptChanges sur les autres lignes pour remettre leurs états à 'Unchanged'.
- Eviter de se faire couillonner par les clés générées.

Je vais détailler ce dernier point. Lorsque vous avez appelé la méthode GetChanges, votre Dataset modifié contenait les valeurs de clés générées par ADO.

Votre Dataset entrant contient lui les valeurs générées par le SGBD. Comme il n'y aura certainement pas identité entre les deux clés, les lignes entrantes vont être ajoutées à la table. Et nous voilà les heureux possesseurs d'un doublon, qui plus est non existant dans la table source.

Il convient dès lors de récupérer les lignes d'origines pour les supprimer. Si vous avez suivi mes conseils, vos lignes ont une valeur de clé négative, et les identifier ne devrait pas poser de problèmes.

Pensez à les supprimer à l'aide de la méthode Remove ou appliquez successivement Delete et AcceptChanges.

Il est possible de modifier le comportement de la méthode Merge en utilisant une de ces surcharges.

public void Merge(Dataset dataSet, bool preserveChanges, MissingSchemaAction missingSchemaAction)

Dans ce cas vous pouvez contrôler plus facilement le comportement de la fusion en préservant ou non les changements.

Notions de liaisons de données

Généralités

Dans cette partie nous allons étudier la liaison de données principalement pour les contrôles de feuilles. Les contrôles liés se présentent en deux familles, ceux qui n'en affichent qu'une (de donnée) et les autres.

La liaison des données aux contrôles est un très vaste sujet que nous allons à peine aborder dans ce cours. Je vous conseille vivement de lire l'aide sur ce sujet.

Les contrôles pouvant être liés utilisent des objets Binding pour se lier avec des données. Un objet Binding défini :

- La propriété du contrôle devant être liée
- La source de donnée (DataView, DataTable…)
- Le chemin de navigation

Prenez l'habitude de travailler sur des objets DataView. Ceux-ci sont fondamentalement faits pour exposer les données. Les modifications faites sur ces objets sont transmises aux objets de données sous-jacents.

Le chemin de navigation peut permettre de spécifier le nom d'un champ ou le nom d'un champ fils au travers d'une relation ou d'autres formes plus complexes. Généralement, on ne manipule pas directement les objets Binding par le code ; ils sont créés par la définition des propriétés des contrôles en mode création / exécution.

Les contrôles qui n'affichent qu'une donnée simultanément (TextBox par ex.) sont appelés contrôles à liaison simple. Ils exposent généralement une propriété DataBindings qui est la collection d'objet Binding du contrôle. Les contrôles à liaison complexe (ListBox, DataGrid) utilisent plutôt des propriétés DataSource et DataMember.

Bien que ce ne soit pas le sujet ici, les contrôles peuvent être liés à autre chose qu'une source de données.

Il n'y a pas grand intérêt à lier un contrôle à liaison simple à une seule donnée. Le but est souvent de pouvoir naviguer dans les enregistrements pour que ces contrôles affichent les données de l'enregistrement en cours. Je vous ai expliqué plus avant la navigation dans un DataView, mais celle-ci n'induit pas le changement d'enregistrement en cours pour les objets Binding. Pour gérer celle-ci, on utilise l'objet CurrencyManager.

CurrencyManager

L'objet CurrencyManager est une couche entre un objet DataView et les contrôles liés. Il prend en charge des notions de position qui permettent aux contrôles dépendant d'afficher l'état d'un enregistrement 'courant'. Attention toutefois avec ce concept. Il ne faut pas le voir comme la gestion de position du moteur de curseur client ADO. L'objet CurrencyManager est lié à une table, pour gérer les CurrencyManager d'un Dataset, on utilise un objet BindingContext.

Propriété

Bindings (BindingCollection)

Renvoie la collection des objets Binding géré par le CurrencyManager

Current (object / DataRowView)

Renvoie l'objet en cours. Dans le cas qui nous intéresse de la liaison aux données d'un Dataset, celui-ci est toujours un DataRowView.

Position (Integer)

Renvoie ou définit la position de l'objet désigné par les contrôles dépendants. C'est par l'intermédiaire de cette propriété que se gère la navigation

Méthodes

Refresh

Force le remplissage des contrôles dépendants.

ResumeBinding, SuspendBinding

Arrête ou reprend la liaison de données.

Evènements

CurrentChange

Se produit lorsque les valeurs liées changent.

MetaDataChanged

Se produit lors de la modification du schéma (Ajout d'une colonne, transtypage)

PositionChanged

Se produit lors du changement de la position.

Navigation

La navigation est assez facile à gérer. Ce code peut paraître lourd car je crée les liaisons de contrôle par le code alors qu'habituellement on le fait directement dans l'environnement de développement.

Notez aussi qu'il faut tester la position dans les boutons Next et Previous.

```
public CurrencyManager VueTable;

    this.Form1.Load += new System.EventHandler(this.Form1_Load);
    …
    private void Form1_Load(object sender, System.EventArgs e){
        OleDbConnection MaConn = new
OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;");
        OleDbCommand MaCommand = new OleDbCommand("SELECT Au_Id, Author,
[year born] FROM Authors", MaConn);
        DataSet dtsBiblio = new DataSet();
        DataView tblAuteurVue;
        OleDbDataAdapter dtAdAuteur = new OleDbDataAdapter();

        dtAdAuteur.SelectCommand = MaCommand;
        dtAdAuteur.Fill(dtsBiblio, "Auteurs");
        tblAuteurVue = new DataView(dtsBiblio.Tables(0));
        this.txtAuteur.DataBindings.Add("Text", tblAuteurVue, "Author");
        this.txtID.DataBindings.Add("Text", tblAuteurVue, "Au_Id");
        this.txtYear.DataBindings.Add("Text", tblAuteurVue, "year born");
        VueTable = (CurrencyManager)(BindingContext(dtsBiblio,
"Auteurs"));
    }

    this.cmdFirst.Click += new System.EventHandler(this.cmdFirst_Click);
    …
    private void cmdFirst_Click(object sender, System.EventArgs e){
        VueTable.Position = 0;
    }

    this.cmdNext.Click += new System.EventHandler(this.cmdNext_Click);
    …
    private void cmdNext_Click(object sender, System.EventArgs e){
        if (VueTable.Position < VueTable.Count – 1) VueTable.Position++;
    }

    this.cmdPrev.Click += new System.EventHandler(this.cmdPrev_Click);
    …
    private void cmdPrev_Click(object sender, System.EventArgs e){
        if (VueTable.Position > 0) VueTable.Position--;
    }

    this.cmdLast.Click += new System.EventHandler(this.cmdLast_Click);
    …
    private void cmdLast_Click(object sender, System.EventArgs e){
        VueTable.Position = VueTable.Count – 1 ;
    }
```

Gestion des exceptions

Jusqu'ici je n'ai pas géré tellement d'exceptions dans les codes que je vous ai fournis. C'est uniquement pour donner un code plus lisible car malheureusement, il faut gérer des exceptions dans un environnement de base de données. Nous allons voir ici les exceptions levées dans ADO.NET.

ConstraintException

Se produit lors d'un conflit autour d'une contrainte. Elle doit être gérée lorsque l'utilisateur saisit les données.

DeletedRowInaccessibleException

Se produit lorsqu'on cherche à agir sur un DataRow supprimé. Rarement utilisé hors de la phase de conception.

DuplicateNameException

Levée si on duplique un nom dans un Dataset. Généralement nom de table, relation ou colonne.

InRowChangingEventException

Jamais utilisée. Pour la déclencher, il faut appeler la méthode EndEdit au sein de l'événement RowChanging.

InvalidConstraintException

InvalidConstraintException est levé lorsque vous appelez les méthodes suivantes de manière incorrecte pendant une tentative de création ou d'accès à une relation.

- DataRelationCollection.Add : peut être levée lorsqu'une relation ne peut pas être créée en se basant sur les valeurs fournies
- DataRowCollection.Clear : peut être levée si ForeignKeyConstraint est appliquée à DataRowCollection
- DataRow.GetParentRow : peut être levée si les colonnes possèdent des types de données différents ou si les tables n'appartiennent pas au même DataSet.

InvalidExpressionException

Se déclenche si vous ajoutez à votre table une colonne contenant une expression invalide. Contient les deux sous erreurs suivantes

SyntaxErrorException

Erreur de syntaxe dans l'expression

EvaluateException

L'expression ne peut pas être évaluée

MissingPrimaryKeyException

Est levée si vous appelez la méthode Contains ou Find de DataRowCollection et que la clé primaire n'est pas définie.

NoNullAllowedException

Celle ci est assez fréquente. Elle se produit dans les cas où on passe une valeur NULL dans une colonne qui ne l'accepte pas. Elle peut être levée avec retard si vous désactivez temporairement les contraintes. Les cas les plus fréquents sont lors de l'appel des méthodes :

- DatarowCollection.Add
- DataRow.EndEdit
- DataRow.ItemArray
- DataTable.LoadDataRow

Mais une opération Fill peut la déclencher aussi.

ReadOnlyException

Comme son nom l'indique, levée sur une tentative de modification d'une valeur en lecture seule.

RowNotInTableException

Déclenchée si vous appelez une des méthodes suivantes sur une ligne qui a été supprimée à l'aide des méthodes Delete ou Remove.

- DataRow.AcceptChanges
- DataRow.GetChildRows
- DataRow.GetParentRow
- DataRow.GetParentRows
- DataRow.RejectChanges
- DataRow.SetParentRow

VersionNotFoundException

Levée lors de l'appel d'une version inexistante de l'objet DataRow. Par exemple l'appel de la version originale d'une ligne ajoutée lève cette exception.

DBConcurrencyException

Celle-là vous la verrez souvent, nous en avons déjà parlée.

Exception du serveur

Toutes les erreurs que le fournisseur rencontre comme erreurs dues au serveur arrive sous la forme OleDbException, OdbcException, SqlException etc…..

Sauf si vous êtes l'administrateur de la source, il ne vous appartient pas de gérer ces exceptions sauf au moment de l'ouverture de la connexion.

Exception Générale

Un grand nombre d'exceptions qui n'appartiennent pas à l'espace de nom system.data peuvent se produire du fait de votre code. Celles-ci proviennent généralement de fautes de types ou d'opération invalides. la plupart du temps, on ne les rencontre qu'en phase de conception.

Manipulation du Dataset

Utiliser une source non sure

Je n'entends pas le terme de source non sure du fait qu'elle présente des risques.

Dans tous les exemples précédents, j'ai construit des Dataset en récupérant des données de SGBD ayant eux aussi des données typées, des contraintes, des relations etc…

Mais il est aussi possible de récupérer des données d'une source qui ne soit pas un SGBD ou un fichier XML. Dans ce cas, la stratégie de récupération se doit d'être différente.

Dans cet exemple nous allons importer les données d'une feuille Microsoft Excel.

Il faut tout d'abord utiliser un fournisseur qui puisse attaquer la cible désirée, dans notre cas MS-Jet.

Pour la suite, il convient de définir si l'on souhaite anticiper les erreurs ou adapter les données à un schéma.

Identifier un schéma

C'est sûrement le plus hasardeux, sauf à utiliser un pilotage d'Excel. En effet rien n'est plus risqué que d'inférer un schéma en fonction de données. Toutefois cela fonctionne si on récupère des données bien formatées. Mais dans le moindre doute, Jet typera la colonne comme une chaîne.

Le code de récupération est assez simple.

```
this.Form1.Load += new System.EventHandler(this.Form1_Load);
    …
    private sub Form1_Load(object sender, System.EventArgs e){
        System.Data.DataSet DSExcel;
        System.Data.OleDb.OleDbDataAdapter dAdExcel;
        System.Data.OleDb.OleDbConnection MaConn;

        MaConn = new
System.Data.OleDb.OleDbConnection(@"provider=Microsoft.Jet.OLEDB.4.0;data
source=C:\\tutoriel\\tuto.XLS;Extended Properties=Excel 8.0;");
        dAdExcel = new System.Data.OleDb.OleDbDataAdapter("select * from
[Test$]", MaConn);
        DSExcel = new System.Data.DataSet();
        dAdExcel.Fill(DSExcel);
        foreach(DataColumn MaCol in DSExcel.Tables(0).Columns){
            MessageBox.Show(MaCol.DataType.ToString);
        }
    }
```

Avec ce code je récupère bien le contenu de la feuille dans mon Dataset, mais le DataAdapter a typé comme il lui semblait bon, bref je n'ai aucun contrôle sur les données entrantes.

Travailler avec FillError

C'est le cas le plus fréquent. Nous avons déjà créé l'objet DataTable et nous voulons insérer les données de notre feuille Excel dans cette table. Pour ce faire nous allons devoir intercepter l'événement FillError.

En quoi celui-ci nous intéresse-t-il ?

Il va permettre de continuer le remplissage malgré les erreurs, et il peut aussi récupérer les lignes ne correspondant pas au schéma afin que l'utilisateur puisse les corriger. Enfin il permet d'ajouter des contraintes sur la table.

```
this.Form1.Load += new System.EventHandler(this.Form1_Load);
    …
    private void Form1_Load(object sender, System.EventArgs e){
        System.Data.DataSet DSExcel;
        System.Data.OleDb.OleDbDataAdapter dAdExcel;
        System.Data.OleDb.OleDbConnection MaConn;

        MaConn = new
System.Data.OleDb.OleDbConnection(@"provider=Microsoft.Jet.OLEDB.4.0;data
source=C:\\tutoriel\\tuto.XLS;Extended Properties=Excel 8.0;");
        dAdExcel = new System.Data.OleDb.OleDbDataAdapter("select * from
[Test$]", MaConn);
        DSExcel = new System.Data.DataSet();
        DataTable TableExcel = new DataTable("Excel");
        TableExcel.Columns.Add("Ident",
System.Type.GetType("System.Int32"));
        TableExcel.Columns.Add("Nom",
System.Type.GetType("System.String"));
        TableExcel.Columns.Add("DateNaissance",
System.Type.GetType("System.DateTime"));
```

```
TableExcel.Columns.Add("Habilité",
System.Type.GetType("System.Boolean"));
        TableExcel.Constraints.Add("pk1", TableExcel.Columns(0), true);
        DSExcel.Tables.Add(TableExcel);
        System.Data.Common.DataTableMapping MonMap = new
System.Data.Common.DataTableMapping();
        MonMap = dAdExcel.TableMappings.Add("Excel", "Excel");
        MonMap.ColumnMappings.Add("Chiffre", "Ident");
        MonMap.ColumnMappings.Add("Nom", "Nom");
        MonMap.ColumnMappings.Add("Date", "DateNaissance");
        MonMap.ColumnMappings.Add("Bool", "Habilité");
        dAdExcel.FillError += new
System.Data.FillErrorEventHandler(this.dAdExcel_FillError);
        dAdExcel.Fill(DSExcel, "Excel");
        MaConn.Close();
        foreach(DataColumn macol in DSExcel.Tables(0).Columns){
            MessageBox.Show(macol.ColumnName.ToString);
        }

        this.DataGrid1.DataSource = DSExcel.Tables(0);
    }

    private static void dAdExcel_FillError(object sender,
System.Data.FillErrorEventArgs args){
        args.Continue = true;
    }
```

Transférer une table

Le principe de l'exercice consiste à transférer une table d'une base de données dans une autre base. Pour cela nous allons pouvoir utiliser le mode déconnecté et le remplissage sans validation. En effet, l'objet DataAdapter expose une propriété AcceptChangesDuringFill qui par défaut est vraie. Celle-ci fait que sur chaque ligne ajoutée dans DataTable, il y a appel de la méthode AcceptChanges. Si je change la valeur d'état de chacune de mes lignes ajoutées va rester à 'Added'. Dès lors, un appel de la méthode Update sur un DataAdapter. Toutefois ceci ne transfère que les lignes. Il faut nécessairement créer la table auparavant. Pour cela, soit-on utilise l'interop COM et un modèle de gestion de structure comme ADOX ou SQL-DMO, soit-on passe par une requête DDL.

J'utilise donc le code suivant avec ADOX :

```
this.Button1.Click += new System.EventHandler(this.Button1_Click);
    …
    private void Button1_Click(System.Object sender, System.EventArgs e){
        ADOX.Catalog MonCat = new ADOX.Catalog();

        MonCat.Create(@"Provider=Microsoft.Jet.OLEDB.4.0;Data
source=C:\\tutoriel\\Essai.mdb");
        ADOX.Table MaTable = new ADOX.Table();
        ADOX.Column MaColId = new ADOX.Column();
        ADOX.Column MaColAut = new ADOX.Column();
        ADOX.Column MaColYear = new ADOX.Column();
        MaColId.Name = "Au_Id";
        MaColId.Type = ADOX.DataTypeEnum.adInteger;
        MaTable.Columns.Append(MaColId);
        MaColAut.Name = "Author";
        MaColAut.Type = ADOX.DataTypeEnum.adVarWChar;
        MaColAut.DefinedSize = 50;
        MaTable.Columns.Append(MaColAut);
```

```
MaColYear.Name = "year born";
        MaColYear.Type = ADOX.DataTypeEnum.adSmallInt;
        MaTable.Columns.Append(MaColYear);
        MaTable.Name = "Authors";
        MonCat.Tables.Append(MaTable);
        MonCat = null;
        OleDbConnection MaConn = new
OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;");
        OleDbCommand MaSelCommand = new OleDbCommand("SELECT Au_Id,
Author, [year born] FROM Authors", MaConn);
        OleDbDataAdapter dtAdAuteur = new OleDbDataAdapter();
        dtAdAuteur.SelectCommand = MaSelCommand;
        dtAdAuteur.AcceptChangesDuringFill = false;
        dtAdAuteur.Fill(dtsBiblio, "Authors");
        OleDbConnection MaNewConn = new
OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\essai.mdb;");
        OleDbCommand MaUpdCommand = new OleDbCommand("Select * From
authors", MaNewConn);
        OleDbDataAdapter dtAdCopie = new OleDbDataAdapter();
        dtAdCopie.SelectCommand = MaUpdCommand;
        OleDbCommandBuilder MonBuild = new
OleDbCommandBuilder(dtAdCopie);
        dtAdCopie.Update(dtsBiblio, "Authors");
        MessageBox.Show("done");
    }
```

Un peu de XML

Nous avons vu que la fabrication d'un schéma n'était pas compliquée. Attention de ne pas confondre schéma et Dataset fortement typé. Un Dataset fortement typé est une classe basée sur un schéma XSD.

L'intégration du XML dans ADO.NET permet de nombreuses opérations. Nous allons en voir trois, souvent utilisées mais il en existe de nombreuses autres.

Synchroniser

La synchronisation se fait toujours entre un Dataset et un XMLDataDocument. Synchroniser signifie que toute modification faite dans l'un est répercutée dans l'autre. L'intérêt est double. Cela permet de gérer la persistance à n'importe quel moment et cela donne accès simultanément à toutes les fonctionnalités d'un Dataset et au service d'un XML. La méthode de synchronisation la plus simple consiste à créer un XMLDataDocument sur un Dataset existant

```
OleDbConnection MaConn = new
OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;");
OleDbCommand MaSelCommand = new OleDbCommand("SELECT Au_Id, Author, [year
born] FROM Authors", MaConn);
OleDbDataAdapter dtAdAuteur = new OleDbDataAdapter();
dtAdAuteur.SelectCommand = MaSelCommand;
dtAdAuteur.FillSchema(dtsBiblio);
dtAdAuteur.Fill(dtsBiblio, "Authors") ;
System.Xml.XmlDataDocument xmlDoc = new
System.Xml.XmlDataDocument(dtsBiblio) ;
```

N'oubliez pas que le schéma est primordial. Un fichier XMLDataDocument offre une vue hiérarchique de données relationnelles. Il existe deux nombreuses façons de rédiger un schéma en XML. Vous devez avoir le schéma le plus complet possible avant de synchroniser.

La deuxième méthode consiste à créer le schéma du Dataset, à synchroniser ce schéma avec le XMLDataDocument puis à charger le document XML Contenant les données. Vous devez vous assurer de la concordance du schéma.

```
OleDbConnection  MaConn = new
OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
Source=C:\\tutoriel\\biblio.mdb;");
```

```
OleDbCommand MaSelCommand = new OleDbCommand("SELECT Au_Id, Author, [year
born] FROM Authors", MaConn);
OleDbDataAdapter dtAdAuteur = new OleDbDataAdapter();
dtAdAuteur.SelectCommand = MaSelCommand;
dtAdAuteur.FillSchema(dtsBiblio);
System.Xml.XmlDataDocument xmlDoc = new
System.Xml.XmlDataDocument(dtsBiblio);
xmlDoc.Load("MonDoc.xml");
```

Enfin il est possible de procéder dans le sens inverse. C'est à dire, charger un document XML dans le XMLDataDocument, créer un Dataset ayant le schéma correspondant et remplir le Dataset en utilisant la propriété Dataset du XMLDataDocument

Annoter le schéma

Les annotations de schéma permettent dans une certaine mesure une manipulation plus aisée de vos Dataset fortement typés. Cela revient à modifier les noms utilisés dans le Dataset et donc ceux apparaissant dans la fonction Intellisense. Les annotations possibles sont les suivantes :

Annotation	Description
typedName	Nom de l'objet.
typedPlural	Nom d'une collection d'objets.
typedParent	Nom de l'objet lorsqu'il y est fait référence dans une relation parente.
typedChildren	Nom de la méthode permettant de retourner des objets d'une relation enfant.
nullValue	Valeur à utiliser si la valeur sous-jacente est DBNull. Consultez le tableau suivant pour les annotations nullValue. La valeur par défaut est _throw.

Il est aussi possible d'annoter nullValue avec les valeurs suivantes :

nullValue	Description
Valeur	Spécifie une valeur à retourner. La valeur retournée doit correspondre au type de l'élément. Par exemple, utilisez nullValue="0" pour retourner 0 pour les champs de type null integer (entier nul).
_throw	Levée d'une exception. Il s'agit de la valeur par défaut.
_null	Retourner une référence null ou lever une exception si un type primitif est rencontré.
_empty	Pour des chaînes, retourner String.Empty, dans les autres cas, retourner un objet créé à partir d'un constructeur vide. Si un type primitif est rencontré, lever une exception.

Pour utiliser des annotations de DataSet typé, vous devez inclure la référence **xmlns** suivante dans votre schéma : `xmlns:codegen="urn:schemas-microsoft-com:xml-msprop"`

Optimisation du code

Les optimisations de codage sont toujours les mêmes, je n'y reviendrai pas. Nous allons parler ici uniquement des règles d'optimisation engendrées par ADO.NET ou les SGBD. N'oubliez pas que votre source de données doit être bien construite. Aucune optimisation d'aucune sorte ne saura vous faire gagner ce qu'une source mal conçue peut vous faire perdre.

Connexion

Ne travaillez pas en connexion maintenue. N'oubliez pas que même dans une architecture client serveur, maintenir une connexion ouverte à un coût. A contrario, ouvrez parfois explicitement pour contrer le fonctionnement par défaut.

Regardons le code suivant :

```
this.OleDbDAAuthors.Fill(this.DtsPubs1.Authors);
this.OleDbDAPublishers.Fill(this.DtsPubs1.Publishers);
this.OleDbDATitAut.Fill(this.DtsPubs1.Title_Author);
this.OleDbDATitles.Fill(this.DtsPubs1.Titles);
```

Ce code n'est pas optimisé. En effet je vous ai dit que le DataAdapter restituait la connexion dans l'état d'origine. Dans mon exemple, la connexion est fermée au début. Lors du premier appel de Fill, le DataAdapter l'ouvre puis la referme. Au second appel, le processus recommence. J'ai donc ouvert et fermé deux fois ma connexion.

Si j'ouvre explicitement ma connexion avant les opérations de remplissage, les appels à Fill ne la referment pas. J'aurais donc ouvert et fermé ma connexion une seule fois.

```
MaConn.Open();
this.OleDbDAAuthors.Fill(this.DtsPubs1.Authors);
this.OleDbDAPublishers.Fill(this.DtsPubs1.Publishers);
this.OleDbDATitAut.Fill(this.DtsPubs1.Title_Author);
this.OleDbDATitles.Fill(this.DtsPubs1.Titles);
MaConn.Close();
```

N'utilisez pas la mise en pool de connexion sur des architectures de type client / serveur.

Ouvrez toujours la connexion explicitement avant de démarrer une transaction

Command

Les objets Command sont des objets légers, très efficaces dans le cadre d'une optimisation.

Utilisez ExecuteNonQuery chaque fois que possible. Utilisez toujours ExecuteScalar quand vous n'avez qu'un résultat à récupérer.

Utilisez 'prepare' sur les requêtes exécutées plusieurs fois.

DataReader

Préférez l'utilisation d'un DataReader plutôt que d'un Dataset si vous n'avez pas besoin de maintenir les données en mémoire.

Utilisez le aussi pour les très gros jeu d'enregistrements.

Fermez toujours le DataReader dès que vous ne vous en servez plus.

Utilisez la récupération typée (GetString, GetInt32….) plutôt que GetValue.

Utiliser l'accès séquentiel (SequentialAccess) lorsque vous n'avez pas besoin de récupérer toute la ligne d'un coup.

Si vous ne lisez pas tous les enregistrements renvoyés, appelez la méthode Cancel de l'objet Command avant de fermer le DataReader.

DataAdapter et DataReader

Evitez d'utiliser le CommandBuilder.

Rapatriez les champs nécessaires et les enregistrements nécessaires. Tout rapatrier est souvent une solution de facilité coûteuse.

Ne récupérez pas le schéma à l'exécution si vous pouvez l'éviter.

Privilégiez la récupération de tables séparées plutôt que de récupérer une table créée sur une jointure.

Récupérez les champs longs avec parcimonie.

Utilisez plutôt des Dataset typés.

Ecrasez plutôt que reconstruire. Si votre table à une clé primaire, re faites un Fill pour remettre à jour votre table. Si vous ne voulez pas perdre vos modifications, utilisez Merge.

Gérez vos TableMappings.

Utilisez des DataView pour les fonctionnalités de recherche ou de filtre. Les DataView créent des index qui améliore grandement la vitesse.

Réduisez les lignes transmises par GetChanges ou Select.

Et surtout, utilisez l'objet plutôt que la position ordinale, la position ordinale plutôt que le nom.

Conclusion

Voilà, nous avons vu les bases de la programmation ADO.NET. Il y aurait encore de nombreux sujets à aborder. Néanmoins vous avez maintenant toutes les connaissances pour développer des applications orientées données. Je vous conseille vivement d'aller fouiller un peu dans l'aide, il y a de nombreux exemples que je n'ai pas traités car ils étaient déjà dedans.

Bonne programmation.