Labo.net SUPINFO

http://www.labo-dotNET.com

ADO.NET

SUPINFO DOT NET TRAINING COURSE

Auteur : Fidèle Tatefo-Wamba et Aleksandar Lukic

Version 1.2 – 19 novembre 2004
Nombre de pages : 37

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

Table des matières

1. Introduction

Quelque soit le type d'application développée, l'accès aux données est un élément important. Le Framework .NET possède une technologie nommée ADO.NET (ADO pour Activex DataBase Object), qui constitue la couche d'accès aux bases de données. Elle permet ainsi aux applications fonctionnant sous .NET d'accéder aux informations stockées dans la plupart des bases de données du commerce.

Anciennement appelé ADO+, ADO.NET est l'évolution directe de ADO. Les modèles présentés par Microsoft furent successivement DAO (Data Access Object), RDO (Remote Data Object) et ADO (Activex Data Object), qui obligeaient les développeurs à apprendre sans cesse à utiliser de nouveaux modes d'accès aux données en fonction du type de base de données utilisée. Par ailleurs, ADO fonctionnait en connexion permanente avec une base de données et ne gérait pas le XML.

ADO.NET fonctionne sur le principe de fournisseurs managés ou fournisseurs codés et est gérer par la plate-forme .NET. Ainsi tous les objets fournis par ADO.NET sont gérés par la CLR sur le principe commun à la compilation (***jus-in-time*** et MSIL), la création et la suppression d'objets (***Garbage Collector***), et à l'exécution d'un programme sous .NET. Ces derniers permettent un accès direct aux bases de données sans qu'il soit nécessaire de connaître les spécificités de chaque base de données et d'identifier les fonctionnalités utilisées par une base de données par rapport à une autre puisqu'il s'agit d'une couche qui uniformise l'accès aux données ; par exemple, certaines fonctions utilisaient des paramètres plus ou moins différents en fonction de la base de données utilisée et il était nécessaire de les connaître pour pouvoir utiliser le fournisseur de données correspondant, ce n'est plus le cas avec ADO.NET.
Il faut noter qu'ADO.NET cohabite avec ADO, ce qui permet de laisser ADO à la disposition des programmeurs à travers des services d'interopérabilité COM de .NET. Il est donc tout à fait possible d'utiliser ADO pour des applications fonctionnant en mode connecté. Néanmoins, des différences profondes existent entre ADO et ADO.NET, notamment en termes de syntaxe, de conception code et de migration.

2. Présentation de ADO.NET

2.1. Configuration nécessaire pour la plate-forme ADO.NET

ADO.NET est fourni avec le Framework .NET et peut être utilisé sous Windows CE/95/98/ME/NT4SP6a/2000/XP. Il est par ailleurs nécessaire d'installer MDAC (Microsoft Data Access Components version 2.6) ou supérieur pour utiliser les fournisseurs de données SQL Server ou OLE DB.

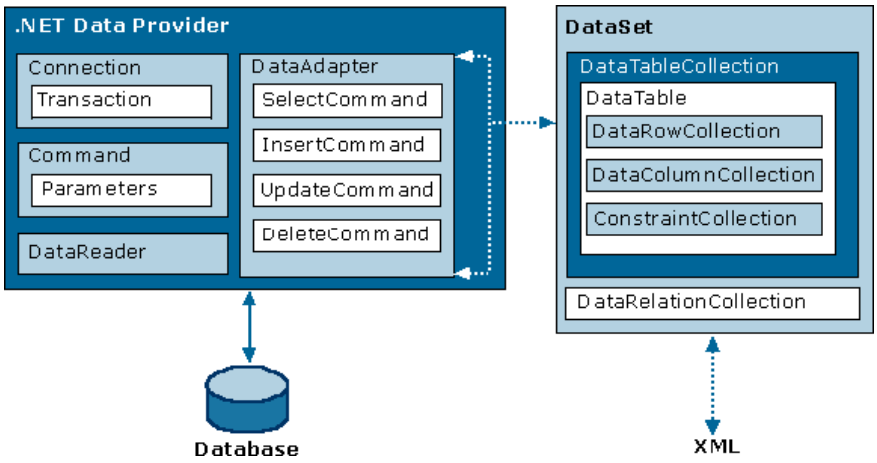
Avant ADO.NET, il existait ADO – ActiveX Data Object. Dans ADO, on manipule des ***Recordsets*** qui font appel aux curseurs côté client ou côté serveur. Ces curseurs sont parcouru ligne par ligne et permettent d'accéder aux données dans une table (colonnes et lignes). Ses curseurs sont gérés par les développeurs grâce notamment à la propriété ***CursorLocation***.
De plus il est impossible d'utiliser ADO à travers les pare-feux. Enfin, l'accès aux données avec ADO nécessite une connexion permanente à la source de données pouvant entraîner des problèmes de performances.

2.2. Vue d'ensemble du modèle ADO.NET

ADO.NET a été conçue avec des optiques un peu différentes :

- Il doit permettre le fonctionnement en **mode déconnecté**. En effet, un des gros problèmes des applications base de données est le goulot d'étranglement lié aux bases de données. Le mode déconnecté permet de charger les données en mémoire et libère ensuite la connexion à la base de données. Cela est possible dans .NET grâce à l'utilisation de l'objet ***DataSet***. L'objet **DataSet** peut être utilisé indépendamment d'un fournisseur de données afin de gérer des données de l'application locale ou des données XML. C'est un des deux modes de fonctionnement possible : en effet, le mode de fonctionnement connecté est toujours disponible.
- Il doit être entièrement **indépendant de la base de données**. En effet, les ***Recordsets*** d'ADO sont plus ou moins dépendant du SGBD et nécessitaient parfois des connaissances particulières sur le SGBD. Maintenant, les objets utilisés dans ADO.NET sont entièrement indépendants.
- Il doit être **intéropérable** avec d'autres systèmes. Pour cela, ADO.NET a été entièrement conçu avec la technologie XML, ce qui lui permet de manipuler des données provenant d'autre chose que de bases de données, comme par exemple un système de fichiers, Active Directory, Index Server, …On dit qu'il uniformise l'interface de programmation pour les classes ***containers*** de données. Quelques soient le type d'application (Web ou Windows) et le type de source de données, les données seront manipulées avec les mêmes classes.
- ADO utilise les ***Variants*** comme type de données (un ***Variant*** est un type de données provenant de Visual Basic, et permettant de stocker tout type de données). Maintenant, ADO.NET utilise un **typage des données fort**, ce qui permet également de faire des vérifications directement à la compilation et non plus à l'exécution comme c'était le cas avec ADO.
- ADO utilise la technologie COM en interne, ce qui ralentit considérablement les temps de traitement (principalement dû au marshalling/unmarshalling de paramètres). ADO.NET a été entièrement réécrit en code .NET et est de ce fait beaucoup plus **performant**.

Pour répondre à toutes ces contraintes, un nouveau modèle de classes a été mis en place dans ADO.NET :



Connected layer :

- Managed Provider
- Accès physique à la base

Diconnected layer :

- Dataset qui cache les données

Figure 1 - Modèle de classe ADO.NET

Ainsi, les objets ADO.NET se divisent en deux catégories : les fournisseurs de données managés et l'objet ***DataSet*** + ses collections.

2.3. Les espaces de noms

Les espaces de nom de la plate forme .NET utilisés pour l'accès aux données sont les suivants :

Espaces de noms	Description
System.Data	Contient les classes de base de l'architecture ADO.NET permettant de construire des composants capables d'administrer les informations en provenance de sources de données multiples.
System.Data.Common	Contient des classes partagées pour les fournisseurs de données managées de .NET. Le fournisseur de données de .NET est une collection de classe qui permet l'accès à une source de données, et donc de la relier à un objet DataSet.
System.Data.SqlTypes	Fournit les classes capables de représenter des types de données en mode natif de SQL Server. Ces classes mettent à la disposition des programmeurs une méthode sécurisée permettant de faire référence à des types de données SQL
System.Data.SqlClient	Contient les classes prenant en charge le fournisseur de données SQL Server de .NET. Ces classes permettent l'accès aux données pour SQL Server 7.0 et supérieur.
System.Data.OleDb	Contient des classes qui prennent en charge le fournisseur de données OLE DB de .NET. Ces classes fournissent un accès managé pour tous les fournisseurs OLE DB pris en charge notamment les SGBD Oracle, Jet, et les versions 6.5 et antérieur de SQL Server

Tableau 1 - Espaces de nom pour l'accès aux données

3. Les composant de ADO.NET

3.1. Les fournisseurs managés

Les fournisseurs managés permettent l'accès aux sources de données (SQL Server, Oracle, …). Ils extraient et présentent les données à travers des classes .NET tels que ***DataReader*** ou ***DataTable***.

Deux fournisseurs managés sont fournis par défaut :

- SQL Server Managed Provider : namespace ***System.Data.SqlClient*** et ***System.Data.SqlTypes***, fonctionne avec SQL Server 7.0 et supérieur. Bien qu'il soit possible d'accéder aux données des versions 7.0 ou supérieur de SQL Server via le fournisseur managé OLE DB, il est conseillé d'utiliser le fournisseur managé de SQL Server pour des raisons de performances. En effet, ce dernier utilise le TDS (***Tabular Data Stream*** ou « flux de données tabulaires »), qui est le protocole de communication natif de SQL Server.
- OLE DB Managed Provider : namespace ***System.Data.OLEDB***, fonctionne avec les OLE DB Providers.
- ODBC Managed Provider : namespace ***System.Data.Odbc***.
- Oracle Managed Provider : namespace ***System.Data.OracleClient***.

Un fournisseur managé comprend les classes ***Command***, ***Connection***, ***DataAdapter***, ***DataReader*** et *Transaction***,** et gère les points suivants :

- Le paramétrage d'une connexion à une source de donnée via l'objet ***Connection***.
- La récupération du flux de données en lecture seule en provenance d'une source de donnée à l'aide de l'objet ***DataReader***.
- La récupération d'un flux d'information en provenance de la source de données et le transfert de ces données à un objet ***DataSet*** pour pouvoir visualiser et mettre à jour ses informations via l'objet de type ***DataAdapter***.
- La synchronisation des mises à jour réalisée dans un DataSet par rapport à la source de données originelle via l'objet ***DataAdapter***.
- La notification des erreurs pouvant survenir pendant la synchronisation des données.

3.1.1. Connection

3.1.1.1. Dans le code

L'objet ***Connection*** permet de créer une connexion à une source de données ou à un fournisseur de données. Il représente une session unique vers une base de données, et est utilisé par des objets tels que l'objet ***Command*** pour effectuer certaines opérations. Par défaut, ADO.NET met à la disposition des développeurs 3 types d'objet ***Connection*** :

- L'objet ***SqlConnection*** pour l'accès aux bases de données SQL Server version 7.0 et ultérieur.
- L'objet ***OdbcConnection*** pour l'accès aux bases de données de type ODBC.
- L'objet ***OleDBConnection*** pour accéder aux bases de données de type OLEDB.

Il faut savoir que ses trois objets sont de simple variante de l'objet **Connection**. En effet, l'interface ***IDbConnection*** est implémentée par ses trois objets. Elle permet donc aux fournisseurs de données de créer leur propre implémentation de l'objet ***Connection***.

Exemple pour une connexion vers SQL Server en utilisant ***SqlConnection***, le provider fourni pour SQL Server :

```
SqlConnection sqlConnection;
sqlConnection = new SqlConnection();
```

```
sqlConnection.ConnectionString = "data source=localhost;"+
      "initial catalog=MyDatabase;integrated security=SSPI";
OleDbConnection oleDbConnection;
oleDbConnection = new OleDbConnection();
oleDbConnection.ConnectionString =
      @"Provider=Microsoft.Jet.OLEDB.4.0;"+
      "Password="""";User ID=Admin;"+
      "Data Source=D:\temp\test.mdb;Mode=Share Deny None;"+
      "Extended Properties="""";Jet OLEDB:System database="""";"+
      "Jet OLEDB:Registry Path="""";"+
      "Jet OLEDB:Database Password="""";"+
      "Jet OLEDB:Engine Type=5;"+
      "Jet OLEDB:Database Locking Mode=0;"+
      "Jet OLEDB:Global Partial Bulk Ops=2;"+
      "Jet OLEDB:Global Bulk Transactions=1;"+
      "Jet OLEDB:New Database Password="""";"+
      "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";
```

Une autre manière de créer d'instancier un objet ***Connection,*** est d'utiliser le concepteur visuel d'application de Visual Studio .NET.

3.1.1.2. A l'aide de Visual Studio

Avant de créer la connexion automatiquement dans le code, il faut d'abord se connecter à la base de données à l'aide de l'explorateur de serveur. Pour cela, dans l'explorateur de serveurs, faire bouton droit sur le nœud "*Data connections*" et choisir la commande "*Add Connection*".

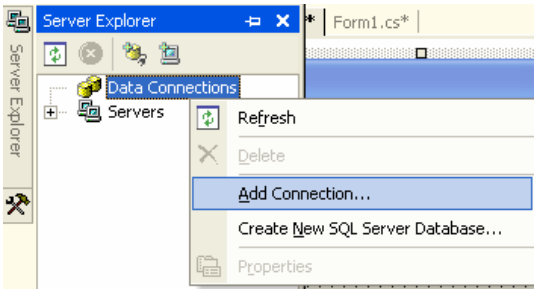


Figure 2 - Connexions à la base de données à l'aide de Visual Studio

Dans la boite de dialogue de figure ci-dessous, il est possible d'indiquer un serveur SQL Server comme source de données ou de choisir le fournisseur managé de la base de données qui sera utilisée. Pour l'exemple actuel, il faut choisir le fournisseur .NET OleDb Jet à partir de l'onglet fournisseur pour se connecter à la base de données Access.

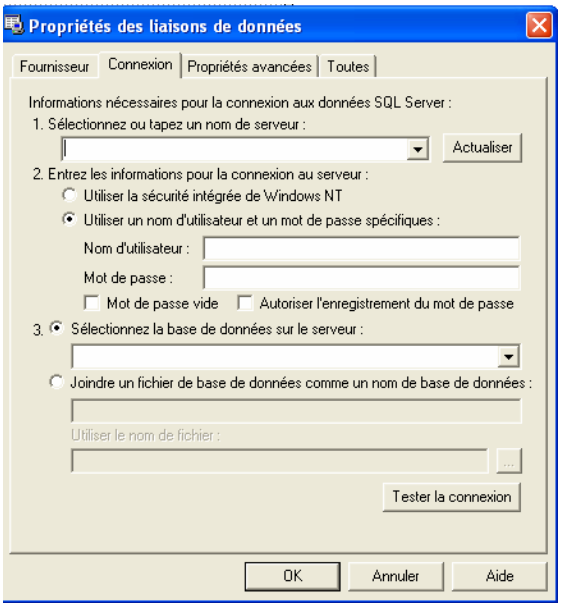


Figure 3 - Choix du serveur SQL Server

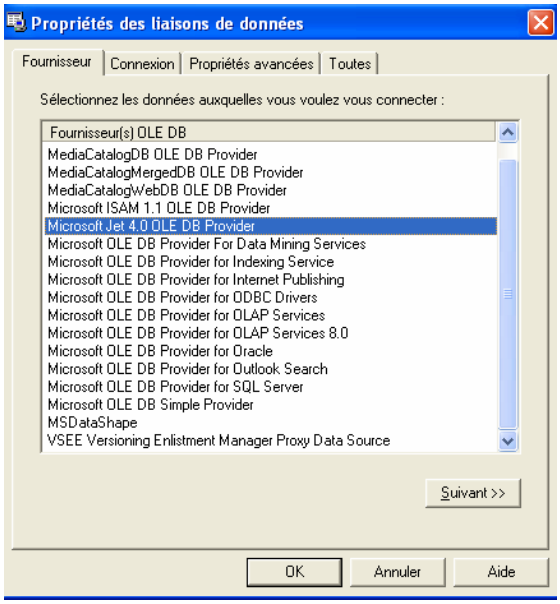


Figure 4 - Choix du moteur JET

En cliquant sur le bouton suivant, l'onglet "*Connexion*" apparaît différemment et propose d'indiquer les informations de connexion à la base de données Access. Le bouton "*Tester*" permet de tester la connexion à la base indiquée dans la boite de texte correspondante.

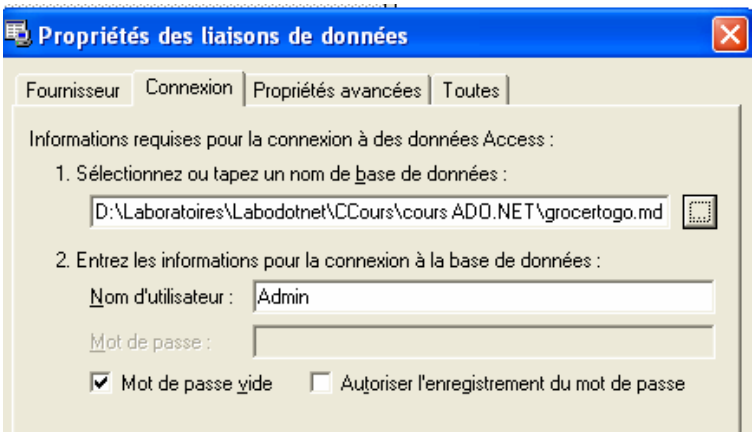


Figure 5 - Information sur le nom d'utilisateur de la base de données

Figure 6 - Test de la connexion

L'onglet "*Propriétés avancées*" permet de modifier les autorisations d'accès à la base de données et l'onglet "*Toute*" permet de modifier une valeur de la chaîne de connexion. Après avoir cliqué sur le bouton "*Ok*", la connexion apparaît dans l'explorateur de serveurs.

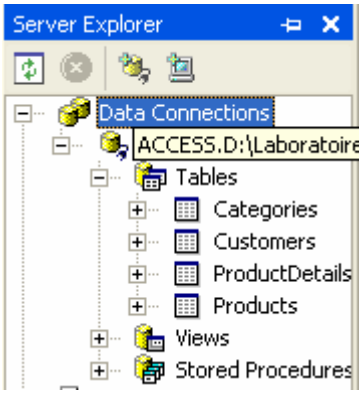


Figure 7 - La base de données dans l'explorateur des serveurs

Une fois la connexion établit, il est possible de visualiser les tables grâce à l'outil "*Visual Data Base Tools*". Maintenant, dans l'onglet "*Data*", il faut glisser-déposer un composant ***OleDbConnection*** dans l'outil d'édition de Visual Studio .NET. Cela à pour effet de créer un objet ***connexion***. Il est possible de modifier ses propriétés notamment son nom et sa chaîne de connexion dans la fenêtre des propriétés.

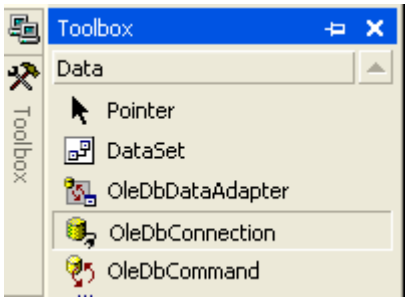


Figure 8 - Contrôle OleDbConnection

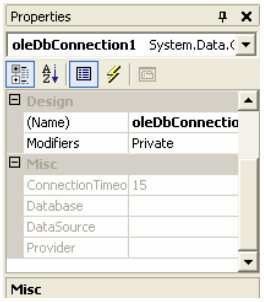


Figure 9 - Propriétés d'un objet OleDbConnection

Pour que la connexion puisse s'effectuer, il faut renseigner la propriété ***ConnectionString*** de l'objet ***Connection***. La chaîne de connexion est l'ensemble des informations nécessaires pour créer une connexion à une source de donnée. Remplir la chaîne de connexion peut se faire directement dans le code (voir paragraphe 3.1.1.1 ***oleDbConnection.ConnectionString***) ou automatiquement par VS.NET dans les propriétés de l'objet récemment crée.

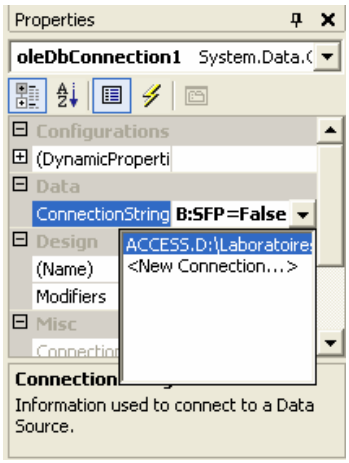


Figure 10 - Création de la chaîne de connexion

Voici quelque unes des autres propriétés de l'objet ***Connection*** :

Nom de la propriété	Description
ConnectionString	Voir plus haut
ConnectionTimeOut	Propriété en lecture seule indiquant la durée en seconde pendant laquelle la méthode ***Open*** attend pour se terminer avant d'abandonner une tentative et de générer une erreur
Database	Propriété en lecture seule retourne une chaîne de caractère indiquant le nom de la base de données employée une fois la connexion ouverte
Driver (ODBC uniquement)	Retourne le nom de la DLL utilisé pour fournir la connexion ODBC.
DataSource	Propriété en lecture seule permettant l'extraction de la source de donnée de la chaîne de connexion telle qu'elle est définie.
Provider (OLEDB uniquement)	Retourne le fournisseur de donnée OLEDB.
ServerVersion	Retourne une chaîne de caractères contenant les informations sur le serveur de base de données qui fait l'objet de la connexion
State	Indique l'état courant de la connexion. Les valeurs possibles pour la propriété ***State*** sont les suivantes : ***Closed*** (base de donnée fermée, état par défaut de l'objet ***Connection*** nouvellement créé) ; ***Connecting*** (lorsque Open est appelée et avant qu'elle ne se termine) ; ***Open*** (lorsque Open s'est terminé avec succès) ; ***Executing*** (Commande en cours d'exécution), ***Fetching*** (données en cours d'extraction) ; ***Broken*** (Objet abandonné).

Tableau 2 - Propriétés de l'objet Connection

Une fois que l'objet ***Connection*** est créé et que la chaîne de connexion est indiquée, il faut ouvrir une connexion à la base de données pour pouvoir y effectuer les manipulations qui nous intéresse. Pour cela, il faut utiliser la méthode ***Open*** de l'objet ***Connexion***. La méthode ***Close*** permet quand à elle de fermer une connexion préalablement ouverte. Il peut être intéressant de tester que la connexion est ouverte avant d'essayer de l'ouvrir pour éviter toute erreur dans l'application.

```
connection.Open();
. . . . . .
connection.Close();
```

Une meilleure méthode consiste à utiliser les clauses ***try finaly*** :

```
Try
{
connection.Open();
. . . . . .
}
finally
{
connection.Close();
}
```

3.1.2. Command

3.1.2.1. Dans le code

Après que la connexion à la source de donnée a été établie, l'objet ***Command*** va permettre d'exécuter des commandes sur une base de données. Chaque fournisseur de donnée de .NET possède sa propre

version de l'objet ***Command***. Ainsi, le fournisseur managé SQL Server possède l'objet ***SqlCommand***, alors que le fournisseur managé OLE DB inclut l'objet ***OleDbCommand***.

L'objet ***Command*** peut aussi bien exécuter des requêtes, des procédures stockées ou des instructions Transact-SQL sur une source de données. Il va référencer dynamiquement l'objet ***Connection*** lors de sa création bien qu'ils soient tous les deux autonomes. Par ailleurs, il est possible de créer l'objet ***Command*** en utilisant la méthode ***CreateCommand*** de l'objet ***Connection***.

Le résultat des données retourné par l'objet ***Command*** est stocké dans l'objet ***DataReader***. Cela ne signifie pas qu'un objet ***Command*** doit absolument retourner des informations.

En plus des objets ***OleDbCommand*** et ***SQLCommand***, .NET possède une interface générique ***IDbCommand***, qui est implémentée par les classes ***OleDbCommand***, ***SQLCommand*** et ***OdbcCommand***.

Constructeurs

Les objets ***SQLCommand*** et ***OleDbCommand*** possèdent un constructeur surchargé permettant de créer une instance des classes correspondantes

```
public SqlCommand();
public SqlCommand( string cmdText);
public SqlCommand( string cmdText, SqlConnection Connection);
public SqlCommand( string cmdText, SqlConnection Connection, SqlTransaction,
Transaction);

public OleDbCommand();
public OleDbCommand( string cmdText);
public OleDbCommand( string cmdText, OldbConnection Connection);
public OleDbCommand( string cmdText, OldbConnection Connection,
OleDbTransaction Transaction);
```

Voici une description des arguments pouvant être fournis au constructeur :

Arguments	Description
cdmText	Instruction SQL, nom de procédure stockée (SQL Server), instruction Transact-SQL (SQL Server). Par défaut, il s'agit d'une chaîne vide.
connection	Objet ***Connection*** à référencer pour utiliser l'objet ***Command***. Indiquer l'objet ***Connection*** correspondant au fournisseur de donnée managé.
transaction	Objet ***transaction***. Il obéit à la même règle que l'objet ***Connection***

Tableau 3 - Argument du constructeur

Quelques propriétés utiles :

Nom	Description
CommandText	Instruction SQL, Transact SQL ou nom de procédure stockée (si la propriété ***CommandType*** est placée à ***StoredProcedure***).
CommandType	Définie ou récupère un flag pour indiquer à l'objet ***Command*** comment est interprétée la valeur contenue dans la propriété ***CommandText***. Les valeurs de cette propriété sont les suivantes : ***Text*** : (par défaut) Indique à l'objet d'interpréter une instruction de Transact-SQL ; ***StoredProcedure*** : Indique que la valeur de ***commandText*** est une procédure stockée ; ***TableDirect*** : Non pris en charge par le fournisseur de managé SQL Server de .NET.
CcommandTimeOut	Définie ou récupère le délai d'attente de l'exécution d'une commande avent que celle-ci ne soit abandonnée et qu'une erreur est renvoyée.
Transaction	Indique ou récupère un objet transaction pour une commande
UpdateRowSource	Indique de qu'elle manière utiliser les résultats d'une commande pour modifier un ***DataRow*** dans un ***DataSet*** losrque la méthode ***Update*** est appelée sur *SqlDataAdapter*. Elle est également requise par l'Interface ***IDbCommand***.
designTimeVisible	Indique si la propriété ***DesignTimeVisible*** est visible dans le control de conception graphique de Visual Studio .NET.

Tableau 4 - Propriétés de l'objet command

Quelques méthodes utiles :

Nom de la méthode	Description
ExecuteReader	Exécute la commande contenue dans la propriété ***CommandText*** sur la connexion assignée et renvoie un objet ***SqlDataReader*** ou ***OleDbDataReader***. Il est possible de contrôler le comportement de l'objet ***DataReader*** utilisé en fournissant un flag ***CommandBehavior***.
ExecuteNonQuery	Exécute une requête SQL sur une connexion et retourne le nombre de ligne affecté sous forme d'entier.

Tableau 5 - Méthodes de l'objet command

Exemple d'utilisation d'un ordre Update:

```
OleDbConnection connection;
OleDbCommand command;
int rowAffected;

connection = new OleDbConnection("...");
command = new OleDbCommand("UPDATE MyTable " +
  "SET MyField = 'MyValue'", connection);
connection.Open();
try
{
  rowAffected = command.ExecuteNonQuery();
}
finally
{
  connection.Close();
}
```

On peut spécifier des paramètres dans la commande, avec un « @ » devant les paramètres. On utilise ensuite la propriété ***Parameters*** de l'objet ***Command*** pour remplir les paramètres.

```
command = new OleDbCommand("UPDATE MyTable " +
  "SET MyField = @MyParam", connection);
command.Parameters["MyParam"].Value = "MyFieldValue";
```

3.1.2.2. A l'aide de Visual Studio

On procède de la même manière que pour l'objet ***Connection***. On glisse et dépose un contrôle ***OleDbCommand*** ou ***SQLCommand***. Puis on modifie ensuite la propriété ***Connection*** dans la fenêtre des propriétés.

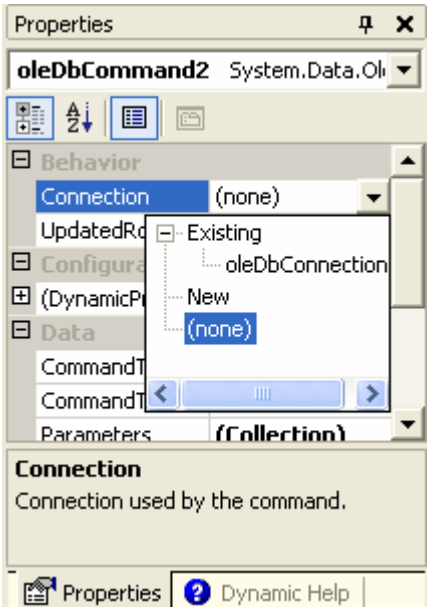


Figure 11 - Choix de l'objet Connection

On peut directement y indiquer la requête SQL à utiliser en choisissant la propriété correspondante (cliquer sur le petit bouton à 3 points). Cela a pour effet d'ouvrir le **constructeur de requête** sur la base de données utilisée par notre objet ***Connexion*** renseigné plus tôt. Il ne reste plus qu'à aller dans le code pour ajouter les méthodes ***Open*** et ***Close*** et à exécuter notre commande.

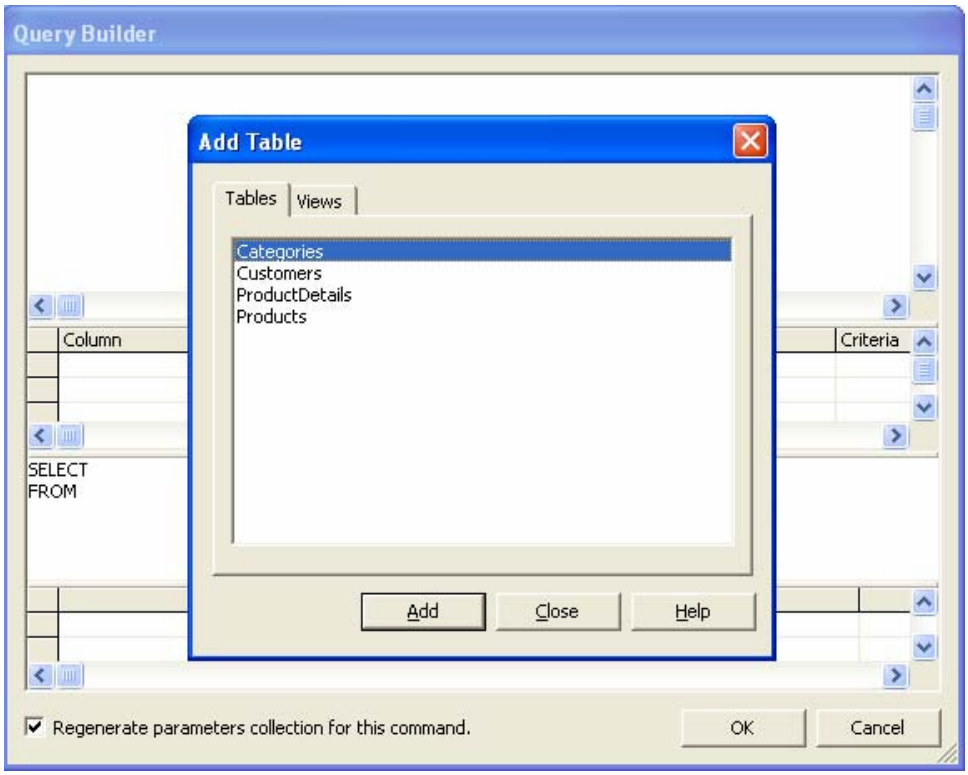


Figure 12 - Constructeur de requêtes

Modifier l'enregistrement "CategoryName" de la table "Categories" pour y ajouter les données passées dans une boite de texte après ouverture de la base de données. Il faut qu'un label indique que celle-ci est ouverte.

Une fois la commande créée, la méthode ***Open*** va permettre d'ouvrir la connexion créée auparavant. Elle est appelée à partir de l'objet ***connection***.

```
connection.Open();
```

Il faut veiller à fermer la connexion à la base de données. Pour le faire, il faut utiliser la méthode ***Close*** de l'objet ***Connection***.

```
connection.Close();
```

3.1.3. DataReader

Sous .NET, il existe 2 mécanismes d'accès aux données d'un entrepôt de données : le ***DataReader*** et le ***DataAdapter***.
Le ***DataReader*** met directement à la disposition des utilisateurs un flux de données rapide, en lecture seul, en avant seulement, ce qui améliore la rapidité de traitement. Il est par contre limité. Voici quelques une de ses caractéristiques :

- L'accès à la base de données se fait par enregistrement. Il ne peut exister qu'un seul enregistrement à la fois dans l'objet ***DataReader***.
- Il est impossible d'effectuer des opérations complexes sur les enregistrements tels que des tris ou des accès directs.
- L'objet ***DataReader*** fonctionne en mode connecté. Aucunes autres actions ne peuvent être effectuées sur la connexion lors de l'accès à la source de données. Il ne faut donc pas oublier de fermer le ***DataReader*** après son utilisation pour libérer la connexion à la base de données.

Tout comme les objets ***Connection*** et ***Command***, il existe aussi une déclinaison différente de l'objet ***DataReader*** pour chaque fournisseur de données de la plate-forme .NET :

- ***SqlDataReader*** pour SQL Server 7.0 et supérieur défini dans l'espace de nom ***System.Data.SqlClient***.
- ***OleDbDataReader*** pour les sources de données OLE DB défini dans l'espace de nom ***System.Data.OleDb***.
- ***OdbcDataReader*** pour les sources de données ODBC défini dans l'espace de nom ***System.Data.Odbc***.

Il n'existe pas de constructeur pour l'objet ***DataReader***. L'instanciation d'un objet ***DataReader*** se fait en utilisant la valeur retournée par l'exécution de la méthode ***ExecuteReader*** de l'objet ***Command***.

```
OleDbCommand command;
command.Connection = oleDbConn;
command.CommandText = ”SELECT * FROM TABLE”;
OleDbDataReader reader =
Command.ExecuteReader(CommandBehavior.ConnectionClose);
```

Voici quelques propriétés de l'objet DataReader :

Nom de la propriété	Description
Depth	En lecture seule, elle retourne un entier indiquant le degré d'encapsulation de la ligne courante dans une requête de modélisation de donnée.
FieldCount	Retourne le nombre de colonnes présentes dans l'enregistrement courant récupéré par l'objet ***DataReader***.
IsClosed	Indique si l'objet ***DataReader*** est fermé
Item	Récupère la valeur de la colonne de données spécifique comme objet .NET. Il existe 2 implémentations surchargées de cette propriété : une pour le nom de la colonne et l'autre pour la position ordinale de la colonne.
RecordsAffected	Indique le nombre de lignes modifiés, ajoutées ou supprimées par l'exécution d'une instruction SQL.

Tableau 6 - Propriétés du DataReader

Quelques méthodes :

Méthode	Description
Close	Définie par l'interface ***IDataReader***, elle permet de fermer le ***DataReader*** afin de libérer la connexion à la base de données
Read	Elle permet la lecture des enregistrements de la table. Elle va déplacer le pointeur de l'objet ***DataReader*** vers l'enregistrement suivant et ensuite va lire les informations de la ligne courante. Elle renvoie ***False*** s'il n'y a pas d'enregistrement suivant à lire, sinon elle renvoie ***True***

Tableau 7 - Méthodes DataReader

Exemple de lecture :

```
OleDbConnection connection;
OleDbCommand command;
OleDbDataReader reader;

connection = new OleDbConnection("...");
command = new OleDbCommand("SELECT * FROM Employee",
  connection);
connection.Open();
try
{
  reader = command.ExecuteReader();
```

```
  try
  {
    while (reader.Read())
      listBox.Items.Add(reader["FieldName"]);
  }
  finally
  {
    reader.Close();
  }
}
finally
{
  connection.Close();
}
```

3.2. Transactions

Une transaction est une opération unique garantissant l'atomicité de séquences d'opération portant sur une base de données. On peut imaginer une opération de transfert de fond dans laquelle le débit effectué sur un compte source et le crédit versé sur un comte cible doivent se succéder parfaitement. Si l'une au l'autre de ses actions échoue, l'opération tout entière est annulée. Ainsi soit tout est exécuté, soit tout est annulé. On parle alors d'atomicité du code. Une transaction bien conçue suit les recommandations ACID : Atomicity, Consistency, Isolation, Durability pour atomicité, cohérence, isolement et durabilité.

Atomicité : Une transaction réussit totalement dans sa tâche ou alors échoue (ne rien faire du tout).
Cohérence : Les données dans une transaction sont dans un état connu quand elle commence et continues à l'être lorsqu'elle se termine.
Isolement : Une transaction ne doit pas voir les modifications apportées sur les données par une autre transaction, elle ne s'occupe que d'elle-même, on dit qu'elle est dans un canal.
Durabilité : En cas de réussite de la transaction, le système garantis que les données validées sont envoyées vers la source de données. Un journal de transaction peut enregistrer des informations de suivit des actions effectuer.

Comme pour les objets étudiés plus haut, il existe 3 classes **Transactions** sous ADO.NET : ***OleDbTransaction***, ***SqlTransaction*** et ***OdbcTransaction*** respectivement pour les transactions OLEDB, SQL Server et ODBC.

L'instanciation d'un objet Transaction se fait en appelant la méthode ***BeginTransaction*** de l'objet ***Connection***. Il est possible d'utiliser l'objet ***Command*** pour opérer sur les données. Pour cela, il faut assigner l'objet Transaction à l'objet ***Command*** à l'aide de sa propriété *Transaction*.

```
myTransaction = MyConnect.BeginTransaction();
mYCommand.Transaction = MyTransaction;
```

Quelques méthodes :

Méthode	Description
BeginTransaction	Crée une transaction imbriquée et retourne une référence sur le nouvel objet ***OleDbTransactionn*** ou ***SqlTransaction***.
Commit	Valide les modifications en attente réalisées sur la base de données à l'intérieur de la transaction courante
Rollback	Annule les modifications en attente effectuées sur la base de données à l'intérieur de la Transaction courante. Elle permet aussi d'annuler les modifications en attente sur la base de données courante depuis le dernier point de sauvegarde.

Tableau 8 - Méthodes de l'objet Transaction

Remarque : Il est possible avec des sources de données SQL Server de créer des points de sauvegarde dans la transaction courante à l'aide de la méthode **Save**.

3.2.1. Exemple de création d'une transaction complète :

```
OleDbConnection oleDbConnection;
OleDbTransaction oleDbTransaction;

oleDbConnection.ConnectionString = "...";
oleDbConnection.Open();
try
{
  oleDbTransaction = oleDbConnection.BeginTransaction();
  try
  {
    // Traitement, requêtes SQL, ...

    oleDbTransaction.Commit();
  }
  catch (Exception ex)
  {
    oleDbTransaction.Rollback();
    throw();
  }
}
finally
{
  oleDbConnection.Close();
}
```

Par contre, il faut spécifier que tous les composants que vous utilisez pour accéder à votre base de données, utilisent cette transaction. Ceci se fait de la manière suivante :

```
myCommand.Transaction = oleDbTransaction;
```

Ainsi, quelque que soit la requête SQL contenue dans ***myCommand***, elle s'exécutera au sein de la transaction.

3.2.2. Les niveaux d'isolement ou IsolationLevel

Il permet d'indiquer comment les transactions vont interagir entre elles. Les niveaux d'isolement définissent des mécanismes de verrouillage qui permettent d'éviter des situations tels que la lecture des données qui ayant été modifiés par une autre transaction n'ont pas été validés ou l'extraction de différents résultats de données lorsque des informations sont relues après une modification des données. Les niveaux d'isolement utilisent un mécanisme de verrou pour protéger l'accès aux informations dans une source de données.

Les niveaux d'isolement sont :

- ***Chaos*** : Aucun accès en écriture n'est autorisé sur les modifications en attente depuis une transaction avec un niveau d'isolement plus élevé.
- ***Unspecified*** : Le fournisseur utilise un niveau d'isolement qui ne peut pas être modifié.
- ***Readuncommitted*** : Toutes les informations peuvent être lues. Aucun verrou n'est placé sur les données. Avec ce niveau, il peut apparaître des lectures incorrectes, des lectures non répétitives, des lectures des lignes fantômes…etc.
- ***ReadCommited*** : Les lectures au cours de l'exécution d'une transaction n'incluent pas les modifications en attente de validation. Les lectures de données modifiées et non validées par une transaction ne sont pas exclues, mais pas les lignes fantômes et les lectures non répétitives.

- ***ReapeatableRead*** : Les autres processus n'ont pas d'accès en écriture sur les données pendant l'exécution d'une transaction, mais les insertions dans les tables sont autorisées. Les lectures incorrectes et les lectures non répétitives sont exclues, mais pas les lignes fantômes.
- ***Serializable*** : les autres processus n'ont pas d'accès en lecture, nie en écriture au cours de l'exécution de la transaction, et les insertions dans la table ne sont pas autorisées. Les lectures incorrectes, les lectures non répétitives et les lignes fantômes sont exclues ;

Exemple :

```
myTransaction = myConnexion.BeginTransaction(IsolationLevel.ReapeatableRead)
```

4. Les procédures stockées

4.1. Présentation

Une procédure stockée est un ensemble d'instructions SQL pouvant s'exécuter de manière atomique. Ceux sont des objets constitués de véritables programmes pouvant recevoir des paramètres, renvoyer des valeurs, être exécutés à distance, posséder leurs propres droits d'accès.
Les procédures stockées possèdent les avantages suivants :

- Elles améliorent les performances des applications : en effet, les procédures stockées sont conservées dans la base de données sous forme d'exécutables. La structure modulaire permet de concevoir de petites unités programmatiques indépendantes qui seront plus rapides à charger en mémoire et partant à exécuter dans une base de données. De plus, elles sont stockées dans le cache mémoire du serveur sous forme compilée lors de leur première exécution (il s'agit du plan d'exécution des procédures stockées), ce qui accroît les performances notamment pour les exécutions suivantes.
- Elles permettent de séparer le code source de l'application du code SQL : l'intérêt est de faciliter la localisation et la modification des requêtes SQL en cas de modification ou de changement de la base de données. On peut imaginer qu'il faille changer de base de données en cours de développement informatique, il ne sera alors pas nécessaire de parcourir tout les code source des différents modules de l'application pour retrouver les requêtes SQL afin de les modifier, il faudra juste modifier les procédures stockées correspondantes. De même, un autre développeur pourra facilement localiser les requêtes SQL d'une application sur laquelle il n'a pas travaillé auparavant.

De nombreuses procédures stockées sont fournis par Microsoft avec SQL Server et sont crées lors de l'installation du server.

Voici les différents cas d'utilisation des procédures stockées :

- Enchaînement d'instructions
- Accroissement des performances
- Sécurité d'exécution
- Manipulation des données système du serveur de base de données
- Mise en œuvre des règles d'entreprise (pour SQL Server notamment)
- Traitements en cascade

4.2. Mise en œuvre

Les procédures stockées acceptent comme beaucoup de langages des paramètres entrées et des paramètres en sorties. Une procédure stockée est capable d'appeler d'autres procédures stockées ou fonctions comme d'être appelées par d'autres programmes.
La création d'une procédure stockée peut se faire de trois façons : à partir du langage de définition des données, à partir de la MMC (Microsoft Management Console) d'Enterprise Manager ou à partir de Visual Studio .NET.

Remarque : La taille maximum d'une procédure stockée est de 128 MB.

4.2.1. Création à partir du LDD (Langage de Définition des Données)

Voici la syntaxe à utiliser pour créer une procédure stockée :

```
CREATE PROC[EDURE] procedure_name [; number]
    [{@ parameter data_type}
```

```
   [VARYING][=default_value] [OUTPUT]
      ][,….n]

   [WITH
  {RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION}]

   [FOR REPLICATION]

   AS sql_statement […n]
  GO
```

Voici une description des arguments utilisés dans cette commande:

Arguments	Description
procedure_name	Nom de la procédure, il doit être unique dans la base de données et se conformer aux règles des identificateurs. Précédé d'un #, la procédure sera temporaire locale, de deux #, elle sera temporaire globale.
number	C'est une valeur entière optionnelle indiquant le numéro d'ordre pour les procédures ayant le même nom.
parameter	Représente un paramètre dans la procédure. Un ou plusieurs paramètres peuvent êtres déclarés dans une procédure. La valeur d'un paramètre déclaré doit être renseignée par l'utilisateur qui se servira de la procédure stockée. Une procédure stockée ne peut contenir plus de 2100 paramètres. Le caractère @ indique qu'il s'agit d'un paramètre.
data_type	Il s'agit du type de donnée du paramètre déclaré. Tous les types de données peuvent être utilisés à l'exception du type table. L'utilisation du type de donnée cursor ne peut être utilisé que sur des paramètres OUTPUT et nécessite l'utilisation des mots clés VARYING et OUTPUT.
Default_value	Variable par défaut du paramètre déclaré.
OUTPUT	Indique s'il s'agit d'un paramètre retourné par la procédure stockée.
n	Indique le maximum des 2100 paramètres pouvant être spécifiés
{RECOMPILE | ENCRYPTION | RECOMPILE,ENCRYPTION }	RECOMPILE : indique que la procédure sera recompilée à l'exécution sans utiliser le cache pour le plan d'exécution de la procédure. Généralement utilisé pour les variables temporaires. ENCRYPTION : Indique le cryptage de l'entrée de la table ***syscomments*** contenant le texte de l'instruction CREATE PROCEDURE. Il permet d'éviter la publication de la procédure dans le cadre de la réplication SQL Server.
FOR REPLICATION	Indique que la procédure stockée doit être exécutée lors de la modification de la table concernée par un processus de réplication. Ainsi, la procédure stockée sera utilisée comme filtre de procédure et ne s'exécutera que lors de la réplication. Cette option ne peut pas s'utiliser avec l'option RECOMPILE.
AS	Indique les actions entreprises par la procédure
Sql_statement	Nombre et type de requête de Transact-SQL qui peuvent être utilisés dans la procédure
n	Indique le nombre maximum de Transact-SQL qui peuvent êtres incluent dans la procédure.
GO	Signale la fin d'un jeu d'instruction

Exemple de code de création d'une procédure stockée :

```
CREATE PROCEDURE GetUsersByID
@UserID int,                                --This is the input parameter
@UserName varchar(50) OUTPUT        --This is the output parameter

AS
-- Get the User name by ID
Select @UserName = Name
From Users                                  -- The table name
WHERE UserID = @UserID

RETURN
GO
```

On peut constater que le paramètre **@UserName** reçoit la le nom de l'utilisateur dont l'ID (**@UserID**) sera indiqué comme paramètre à la procédure stockée **GetUserById**. Le paramètre **@UserName** sera par ailleurs renvoyé par la procédure stockée.

4.2.2. Création à partir de Entreprise Manager

Pour cela, à partir de l'outil **Entreprise Manager** dans le dossier Microsoft SQL Server des programmes du **menu démarrer**, il faut dérouler le dossier **Bases de données**, puis dérouler la base de données concernée par la manipulation, faire bouton droit sur le nœud Procédures stockées (Stored procedure) et choisir la commande **Nouvelle Procédure stockée** comme illustrée ci-dessous.

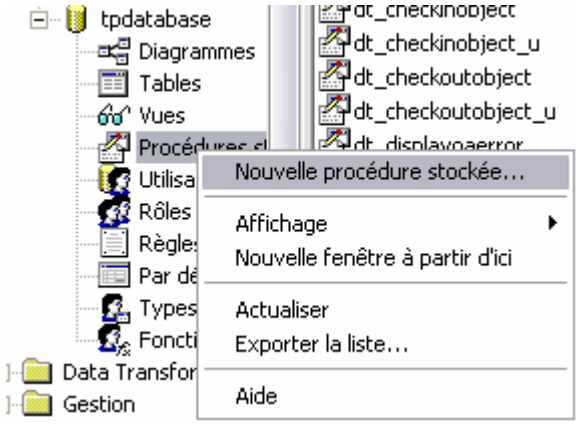


Figure 13 - Création d'un procédure stockée dans Enterprise Manager

4.2.3. Création à partir de VS.NET DataBase Tools

Il faut pour cela qu'une connexion à la base de données ait été effectuée à partir de l'explorateur de serveur. Si cette condition est vérifiée, il faut juste dérouler le nœud correspondant à la base de données qui possèdera les procédures stockées, sélectionner le nœud "**Stored Procedures"**. Cliquer avec le bouton droit de la souris sur ce nœud et choisir la commande "**New Stored Procedure"** comme illustré ci-dessous.

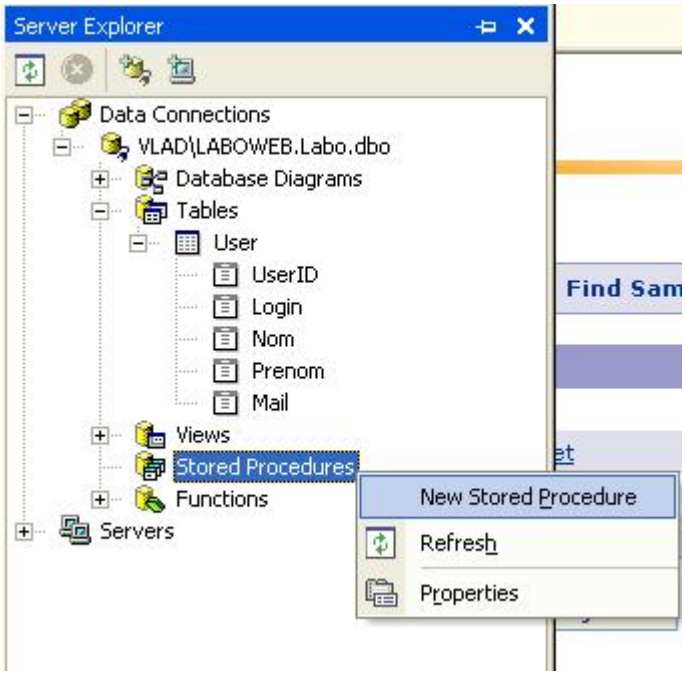


Figure 14 - Création d'une procédure stockée dans Visual Studio

VS.NET crée alors un onglet d'édition supplémentaire dans sa fenêtre principale destiné à l'écriture du code de la procédure stockée nouvellement crée.

```
1  CREATE PROCEDURE GetUsers
2  /*
3      (
4          @parameter1 datatype = default value,
5          @parameter2 datatype OUTPUT
6      )
7  */
8  AS
9      /* SET NOCOUNT ON */
10     RETURN
11
```

Figure 15 - Création de procédure stockée dans Visual Studio

Une fois la commande enregistrée lancée, on peut constater que Visual Studio.NET modifie le code de la procédure stockée.

```
1  ALTER PROCEDURE GetUsers
2  /*
3      (
4          @parameter1 datatype = default value,
5          @parameter2 datatype OUTPUT
6      )
7  */
8  AS
9      /* SET NOCOUNT ON */
10     RETURN
11
```

Figure 16 - Modification de procédure stockée dans Visual Studio

4.3. Implémentation des PS dans le code C#

4.3.1. Création de la commande et appel de la procédure

Une fois une procédure stockée crée sur le serveur de base de données, il est facile d'appeler celle-ci dans le code C#. Il faut avant tout s'assurer de disposer d'un objet de la classe ***Command*** (objets ***SqlCommand*** ou ***OleDbCommand***). Pour pouvoir utiliser une procédure stockée, il faut indiquer le nom de la procédure stockée comme paramètre du constructeur de l'objet et indiquer à la propriété ***CommandType*** de cet objet qu'il s'agit d'une procédure stockée (utiliser la valeur de l'énumération ***CommandType***).

Exemple :

```
SqlCommand  com= new SqlCommand("GetUsers");
com.CommandType = CommandType.StoredProcedure
```

Dans l'exemple ci-dessus, le nom de la procédure stockée est ***GetUsers***.
Il est aussi possible de procéder de la sorte :

```
SqlCommand com = new SqlCommand();
com.CommandType = CommandType.StoredProcedure
com.CommandText="GetUsers"
```

4.3.2. Passage et récupération des différents paramètres

Pour passer des paramètres à la procédure stockée, il faut créer un objet de la classe ***SqlParameter*** en indiquant comme argument du constructeur, le paramètre créé dans la procédure stockée et le type de ce paramètre, en second argument. La propriété ***Value*** de l'objet de type ***SqlParameter*** va indiquer le champ concerné par le paramètre renseigné plus haut. La méthode ***Add*** permet d'ajouter l'objet créé à la collection des paramètres (***SqlParameterCollection***) de la commande.

Exemple de passage de paramètres:

```
SqlParameter parUserID= new SqlParameter("@UserID",SqlDbType.Int);
parUserID.Value = userID;
com.SelectCommand.Parameters.Add(parUserID);
```

La récupération des informations passées par le paramètre de sortie se fait exactement comme s'il s'agissait d'une requête SQL de sélection à l'aide d'un ***DataReader*** ou d'un ***DataSet***.

Exemple de récupération de paramètres avec un ***DataSet*** :

```
DataSet myDS = new DataSet();
……
SqlParameter parUserID= new SqlParameter("@UserID",SqlDbType.Int);
parUserID.Value = userID;
com.SelectCommand.Parameters.Add(parUserID);
try
{
com.Fill(myDS);
}
finally
{
      myConnection.Close();
}
```

5. DataAdapter

Avant de commencer éclaircissons un point de syntaxe. Dans ce cours il sera question de ***DataAdapter***, alors que les composants que vous manipulez s'appelleront ***SqlDataAdapter*** ou ***OleDbDataAdapter***. Ces deux composants sont identiques, mais comme vous l'aurez sûrement compris, un est optimisé pour une connexion OleDb (Access par exemple) et l'autre pour une connexion SQL (SQL Server par exemple). Nous pouvons à présent continuer.

Le ***DataAdapter*** contient en fait un ensemble de méthode permettant l'accès à la base de données. On y trouve quatre instances de la classe ***Command*** :

- ***SelectCommand*** : permet de sélectionner des données dans la base
- ***InsertCommand*** : permet d'insérer des données dans la base
- ***UpdateCommand*** : permet de mettre à jour des données dans la base
- ***DeleteCommand*** : permet de supprimer des données dans la base

Il y a plusieurs constructeur, cependant le plus utilisé prend en premier paramètre une chaîne de caractère qui représente une requête SQL et qui initialisera la command ***SelectCommand***. Le second paramètre est une instance de la classe ***Connection***.

Voici un exemple d'initialisation :

```
string query = "SELECT * FROM MA_TABLE";
OleDbDataAdapter MyDataAdapter = new OleDbDataAdapter( query ,
oleDbConnection);
```

On suppose bien sur que la connexion ***OleDbConnection*** a été initialisée.

Voici le fonctionnement :

Dans la partie Data de la ToolBox se trouve les composants ***DataAdapter***. Double cliquez sur l'un d'eux. Un assistant va s'afficher :

Figure 17 - Assistant de création de DataAdapter

Cliquez sur "Next" pour arriver sur une fenêtre vous permettant de spécifier la connexion à utiliser. Vous pouvez en créer une ou en utiliser une déjà créé :

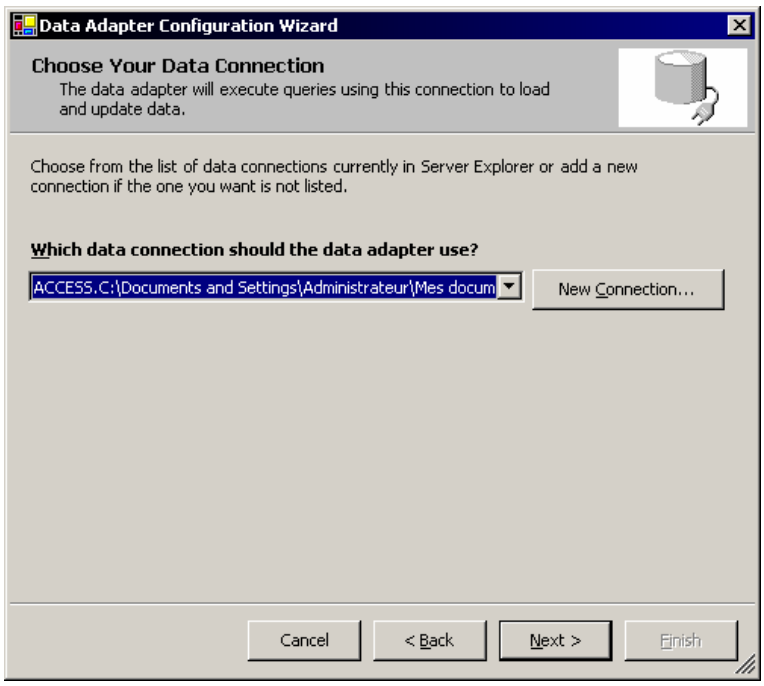


Figure 18 - Choix de la connexion

Cliquez sur "Next" pour afficher la fenêtre permettant de spécifier le type de requêtes à créer. Suivant le type de base à laquelle vous vous connectez, vous aurez plus ou moins de choix possible. Dans notre cas, nous utilisons une connexion à une base Access, nous ne pouvons donc pas créer de procédure stockée :

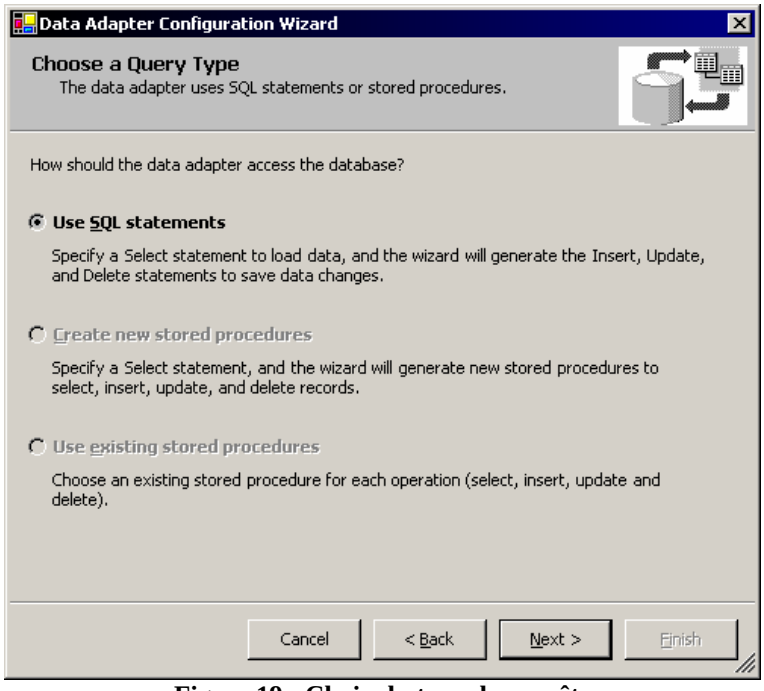


Figure 19 - Choix du type de requête

La fenêtre qui suit vous permet de spécifier les tables et champs à utiliser, soit en créant vous-même la requête soit en passant par le Query Builder.

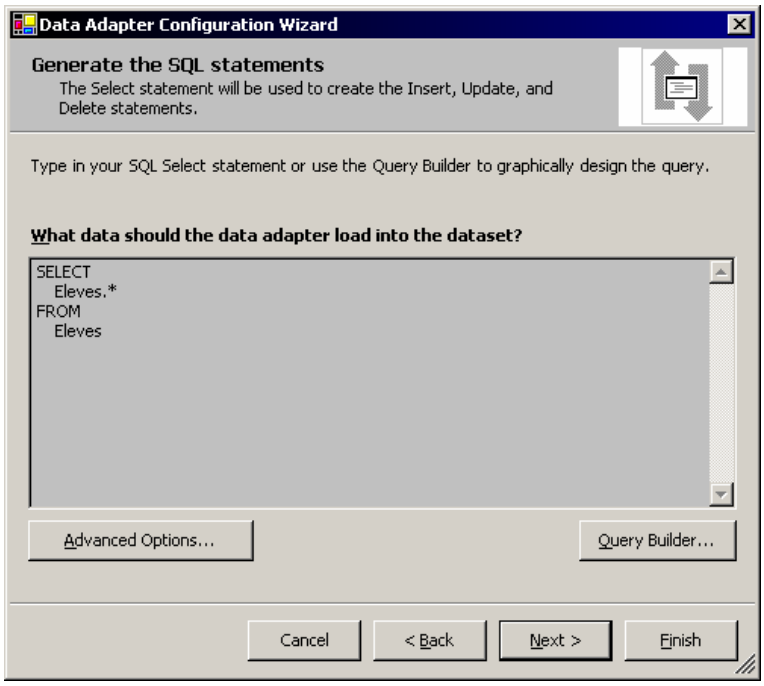


Figure 20 - Utilisation Query Builder

La dernière fenêtre vous indique ce que l'assistant à réussi ou pas à générer. Cependant vous devez savoir que l'assistant à ces limites. En effet, si vos tables ne contiennent pas de clé primaire, il ne pourra générer que la requête SELECT. De la même façon, si vous avez choisis des données de plusieurs tables, il ne pourra générer aucune requête.

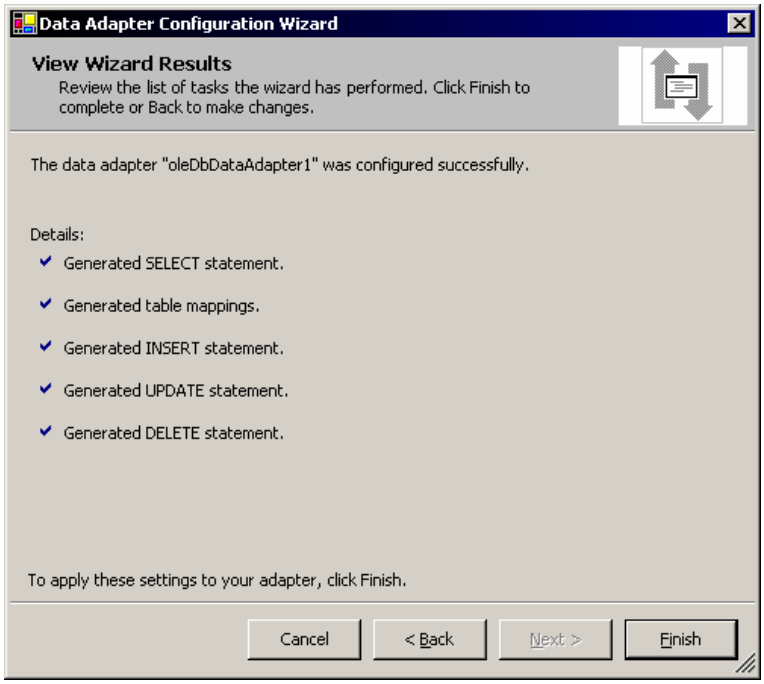


Figure 21 - Fin de configuration du DataAdapter

Vous avez donc créez un ***DataAdapter*** avec le designer. Mais les avantages de cette méthode ne s'arrêtent pas là. Dans la fenêtre propriétés, vous pouvez trouver tout en bas quelques liens très intéressant :

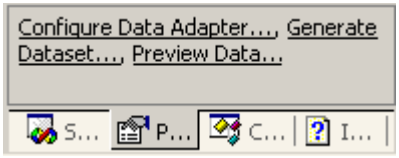


Figure 22 - Liens intéressant de la fenêtre Propriété du DataAdapter

"Configure Data Adapter …" : relance l'assistant pour modifier le ***DataAdapter*** déjà créé
"Generate DataSet" : va automatiquement générer un ***DataSet*** Typé (voir paragraphe 8)

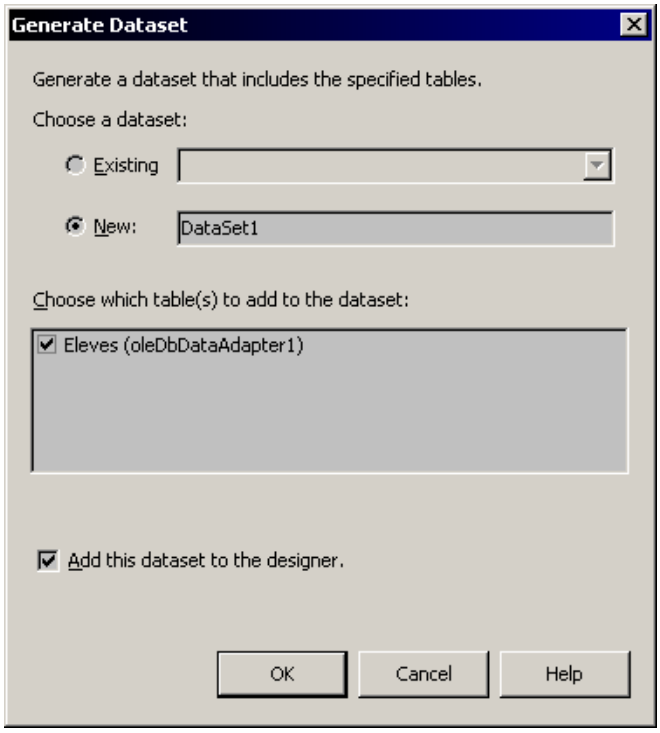


Figure 23 - Génération du DataSet

"Preview Data" : prévisualisation des données obtenues grâce au ***DataAdapter***

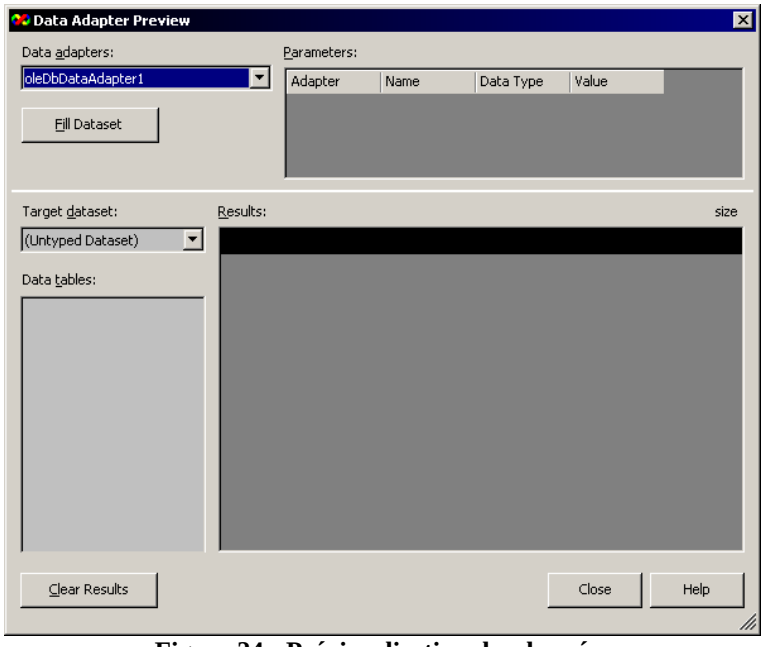


Figure 24 - Prévisualisation des données

Deux autres méthodes très intéressantes peuvent être citées. Il s'agit de ***Fill*** qui permet de remplir un ***DataTable*** ou de créer un ***DataTable*** dans un ***DataSet*** suivant le paramètre fourni. La deuxième méthode est ***Update***, qui permet de mettre à jour la base de données automatiquement. Le fonctionnement de cette méthode sera détaillé un peu plus loin dans ce cours.
Le ***DataSet*** est une représentation en mémoire de données. Lorsqu'il contient des données provenant d'une base de données, il permet de travailler en mode déconnecté. Dans ce cas, on a souvent besoin des données de la table que l'on veut traiter, mais également des données provenant d'autres tables de la base de données lorsqu'elles sont liées par des contraintes.
Le ***DataSet*** doit donc fournir des informations complémentaires quant à la structure du modèle de données, comme par exemple les relations entre les tables, les clés primaires, les types des colonnes… Ces informations nous permettront de pouvoir travailler efficacement sur des données, et ce en mode déconnecté.

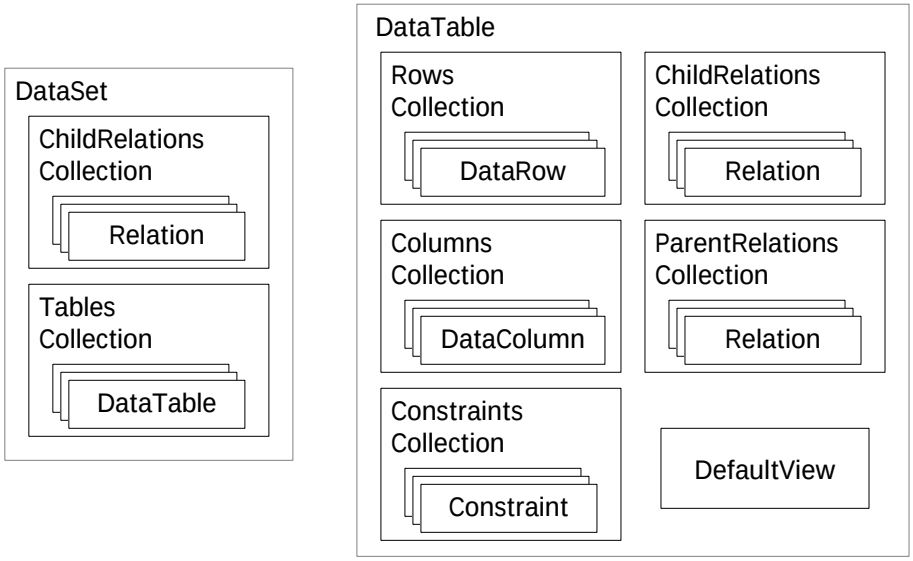


Figure 25 - Informations contenues dans un DataSet

Le ***DataSet*** est indépendant du modèle des données et indépendant de la source de données. C'est pourquoi on peut y stocker des données provenant de n'importe quelles bases de données (SQL Server, Oracle, Access, …), d'un fichier XML sur disque.
On peut tout à fait créer un ***DataSet*** de toute pièce, spécifier des tables, des champs, des contraintes… et utiliser ensuite ce ***DataSet***.

5.1. Propriété Tables : tableau de DataTable

La propriété ***Tables*** permet d'obtenir une collection de ***DataTable***. Un ***DataSet*** contient une ou plusieurs ***DataTable*** (comme vous pouvez le voir sur le schéma ci-dessus).
Une ***DataTable*** est donc une représentation en mémoire de données. Comme vous avez pus le voir dans la partie sur les ***DataAdapter***, ces données peuvent être chargées par la méthode ***Fill***.
Cependant vous pouvez également créer une ***DataTable*** de toutes pièces en créant les lignes et les colonnes et en insérant les données vous-même.

5.2. Propriété Columns : tableau de DataColumn

La propriété ***Columns*** d'une ***DataTable*** permet d'obtenir une collection de ***DataColumn***. Ce qui implique bien entendu qu'une ***DataTable*** peut avoir plusieurs ***DataColumn***.
DataColumn contient des informations sur le type du champ. En voici quelques unes :

- ***AllowDbNull*** : indique si les données contenues dans la colonne peuvent avoir la valeur ***null***.
- ***AutoIncrement*** : indique si la colonne doit incrémenter automatiquement la valeur de la colonne lors de l'insertion d'une nouvelle ligne.
- ***Caption*** : permet de donner ou d'obtenir le nom de la colonne.

- ***DataType*** : permet de d'obtenir ou de définir le type des données de la colonne.
- ***DefaultValue*** : obtient ou défini la valeur par défaut de la colonne.

Comme vous pouvez le remarquer, on peut facilement faire une correspondance entre les ***DataTable*** et les tables des bases de données.

5.3. Propriété Rows : tableau de DataRow

La propriété ***Rows*** permet d'obtenir une collection de ***DataRow***. Une ***DataRow*** représente une ligne d'une ***DataTable***. Elle permet d'accéder aux valeurs d'une ***DataTable*** en spécifiant un nom de colonne ou un numéro de colonne.

Vous en connaissez assez maintenant pour pouvoir accéder à des données contenues dans une DataTable. Voyons donc un exemple :

```
DataTable dtbEmployee;

dtbEmployee = dataSet.Tables["EMPLOYEE"];
foreach (DataRow row in dtbEmployee.Rows)
listBox.Items.Add(row["EMP_NAME"].ToString());
```

Dans cet exemple, on va chercher chaque ligne de la table "EMPLOYEE" contenue dans le ***DataSet dataset***. On ajoute ensuite le contenu de la colonne "EMP_NAME" dans une ***ListBox listBox***.

Remarque : Comme vous pouvez le voir dans l'exemple ci-dessus, pour accéder à la valeur d'une colonne on utilise le nom du champ. On aurait pu utiliser l'index de la colonne mais il est très important de préférer la première technique à la seconde. En effet, si le nombre de colonne de la table est amené à changer, il faudra alors revoir tout le code qui utilise cette table.

5.4. Propriété Constraints : tableau de Constraint

La propriété ***Constraints*** permet d'obtenir une collection de ***Constraint***. Pour rappel, les contraintes permettent de maintenir l'intégrité d'une base de données.

Les ***DataTable*** supportent 2 types de contraintes :

- Unique Key Constraints (ou clé primaire) avec la class ***UniqueKeyConstraint*** qui permet de vérifier que les valeurs d'une table sont uniques. Si la propriété ***EnforceConstraints*** du ***DataSet*** est ***True***, la violation d'une contrainte va générer une exception.
- Foreign Key Constraints (ou clé étrangère) avec la class ***ForeignKeyConstraint*** qui permet de spécifier comment agir lorsque une donnée dépendante de deux ou plusieurs tables différentes est supprimées. Faut-il la supprimer dans les autres tables, la mettre à ***null*** ou mettre une valeur par défaut ?

Les contraintes permettent la validation des données sans nécessité de contacter la base de données afin de maintenir une déconnexion totale avec la base de données.

Ces deux relations sont crées automatiquement lors de la création d'un objet ***DataRelation***, mais peuvent être créées séparément.

5.5. Propriété Relations : tableau de DataRelation

La propriété ***Relations*** de la class ***DataSet*** permet d'obtenir une collection de ***DataRelation*** ou ***DataRelationCollection***.

Une ***DataRelation*** permet de lier deux tables du ***DataSet*** au travers des colonnes ou ***DataColumn*** de celles-ci. Ces relations permettent de représenter les relations liant les clés étrangères et les clés primaires.
On retrouve comme d'habitude des notions de base de données toujours pour rester le plus proche de ces dernières et ainsi avoir une représentation identique en mémoire lors du fonctionnement en mode déconnecté.

Voici une fonction qui crée une relation entre la colonne "ProductID" des tables "ProductDetails" et "Products" :

```
private void CreateRelations()
{
      DataColumn parentColumn;
      DataColumn childColumn;
      parentColumn = DS.Tables["Products"].Columns["ProductID"];
      childColumn = DS.Tables["ProductDetails"].Columns["ProductID"];
      DataRelation relCustOrder;
      relCustOrder = new DataRelation("ProductIDRelation", parentColumn,
childColumn);
      DS.Relations.Add(relCustOrder);
}
```

Comme vous pouvez le voir, le code est très détaillé. On crée tout d'abord deux colonnes qui correspondent à celles que l'ont veut lier. On crée ensuite une ***DataRelation*** qui aura pour nom "ProductIDRelation". Et finalement, on ajoute cette relation au ***DataSet***.

6. DataBinding

Le ***DataBinding*** consiste à prendre les données d'une source que l'on appelle généralement « provider » (***DataTable***, ***DataView***, ***ArrayList***…) et de les placer, par un simple appel de méthode, dans un contrôle qui est appelé « consumer » (***DataGrid***, ***DataList***, ***DropDownList***…).
La relation entre Provider et Consumer est appelée Binding.

6.1. Le DataBinding dans le code

Les composants visuels de .NET possédant une propriété ***DataSource*** peuvent se connecter à un ***DataSet***, ***DataTable*** ou ***DataView*** pour afficher des données.
Voyons dans un exemple comment faire avec une ***DataGrid*** sachant que avec les autre contrôle la technique est exactement la même :

```
DataGrid1.DataSource = dataSet;
DataGrid1.DataSource = dataSet.Tables["EMLOYEE"];
```

On voit donc ici que l'on peut connecter la ***DataGrid*** directement à un ***DataSet*** ou bien à une table de ce ***DataSet***.

Attention : En ce qui concerne certains contrôles, comme un ***ComboBox*** ou une ***ListBox***, l'opération se révèle un peu plus difficile. En effet, étant donné que ces contrôles possèdent un nombre limité de colonnes, il vous faudra passer par une étape intermédiaire qui consiste à construire un ***ArrayList*** contenant les éléments que vous souhaitez mettre dans la ***ComboBox*** et seulement après procéder au Binding.

Voici un exemple de code :

```
ArrayList arList = new ArrayList();
string Query = "Select LastName, FirstName from Employees";
DataTable dtEmp = new DataTable();
OleDbDataAdapter daEmp = new OleDbDataAdapter(Query, oleDbConnection);

daEmp.Fill(dtEmp);
foreach(DataRow dr in dtEmp.Rows)
      arList.Add(dr[0].ToString() +", "+dr[1].ToString());
comboBox1.DataSource = arList;
```

On suppose que la connexion ***OleDbConnection*** a été initialisée autre part dans le programme.

Il existe une autre méthode plus élégante mais qui ne vous permet cependant pas d'afficher deux données (nom, prénom par exemple) dans une colonne. La voici illustrée par l'exemple qui suit :

```
string Query = "Select LastName, FirstName from Employees";
DataTable dtEmp = new DataTable();
OleDbDataAdapter daEmp = new OleDbDataAdapter(Query, oleDbConnection);

daEmp.Fill(dtEmp);
dtEmp.Columns[0].ColumnName = "LastName";
dtEmp.Columns[1].ColumnName = "FirstName";
comboBox1.DataSource = dtEmp;
comboBox1.DisplayMember = "LastName";
comboBox1.ValueMember = "FirstName";
```

Comme vous pouvez le voir, cette technique consiste à utiliser le nom d'une colonne. L'avantage par rapport à la technique précédente, est que vous pouvez ici facilement mettre une valeur à la fois dans les propriétés ***DisplayMember*** et ***ValueMember***.

6.2. Le DataBinding dans le designer

Dans la section précédente, on a pu voir comment manipuler le DataBinding dans le code. Cependant, VS.NET permet d'arriver aux mêmes résultats en passant par le designer et en n'utilisant que la souris.
Pour cela, il vous suffit de créer une connexion et un ***DataAdapter*** dans le designer (voir paragraphe 3.1.1.2).
Vous devez en suite générer un ***DataSet*** avec l'aide du ***DataAdapter***.
Il ne vous reste plus qu'à « binder » le contrôle dans lequel vous désirez afficher les données avec l'aide des propriétés ***DataSource*** et ***DataMember*** du contrôle en question (voir ci-dessous).

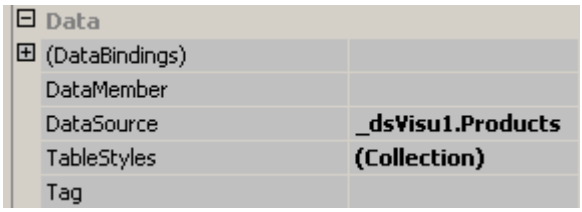


Figure 26 - Liaison du contrôle à la source de donnée

6.3. DataView

DataView permet de créer une vue personnalisée d'une ***DataTable***. Cela veut dire que l'on peut n'afficher que certains éléments de la ***DataTable*** et en omettre d'autre sans pour autant la modifier. On dit que l'on filtre.
Il n'est pas possible d'enlever ou d'ajouter une ou des colonnes de la table source.
Un ***DataView*** a les propriétés : ***Sort***, ***RowFilter*** et ***RowStateFilter***. Ces propriétés permettent de customiser la vue sur la ***DataTable***. L'avantage des ***DataView*** est de pouvoir être utilisé en tant que Provider dans un DataBinding.

Voyons quelques propriétés de la ***DataView*** :

- ***AllowDelete*** : Obtient ou définit un booléen indiquant si l'on peut supprimer des données. Ces suppressions qui seront répercutées sur la ***DataTable*** source.
- ***AllowEdit*** : Obtient ou définit un booléen indiquant si l'on peut modifier des données. Ces modifications seront répercutées sur la ***DataTable*** source.
- ***AllowNew*** : Obtient ou définit un booléen indiquant si l'on peut ajouter des données. Ces ajouts seront répercutés sur la ***DataTable*** sources.
- ***RowFilter*** : Obtient ou définit une expression servant de filtre.
- ***RowStateFilter*** : Obtient ou définit l'état des lignes utilisé dans la ***DataView***. Ces filtres peuvent par exemple être ***Deleted***, ***CurrentRows***, ***OriginalRows*** …

Exemple :

```
private void CreateDataView()
{
      DataView dataView = new DataView(ds.Tables["ProductDetails"], "Grams =
'0g'", "ProductID", DataViewRowState.CurrentRows);
      dataView.AllowEdit = true;
      dataView.AllowNew = true;
      dataView.AllowDelete = true;
}
```

On ici que l'on crée une ***DataView*** qui va afficher les données de la table "ProductDetails" dont le poids est égal à 0g. On pourra éditer, ajouter et supprimer des lignes. Ces actions seront bien sur prises en compte dans la ***DataTable*** source.

7. Mise à jour des données

7.1. En utilisant Command

La plus simple des façons de mettre à jour des données est d'utiliser les classes ***OleDbCommand*** ou ***SqlCommand***. En effet, il suffit de créer un objet ***Command***, de spécifier votre requête SQL de mise à jour (INSERT, UDPATE ou DELETE), d'affecter des valeurs aux paramètres s'il y en a, et ensuite d'exécuter la commande.

Exemple d'exécution d'un ordre UPDATE :

```
OleDbConnection connection;
OleDbCommand command;
int rowAffected;

connection = new OleDbConnection("...");
command = new OleDbCommand("UPDATE MyTable " +
  "SET MyField = 'MyValue'", connection);
connection.Open();
try
{
  rowAffected = command.ExecuteNonQuery();
}
finally
{
  connection.Close();
}
```

L'inconvénient de cette méthode est qu'elle s'applique très mal au mode déconnecté. En effet, dans ce cas le développeur doit être au courant des données à mettre à jour. Dès que les données sont complexes et importantes en nombre, le travail devient trop grand et l'intérêt de travailler en mode déconnecté n'est plus vérifié.

7.2. En utilisant DataSet et DataAdapter

Dans ce cas là, la méthode change, car nous travaillons en mode déconnecté et de ce fait, nos modifications ne sont prises en compte qu'en local. Il faut donc les « rapatriées » dans la base de données.

La mise à jour des données s'effectue en deux étapes :

- Effectuer les changements sur les ***DataRows*** du ***DataTable*** (sans connexion à la base de données).
- Transférer les changements du ***DataTable*** vers la data source en une seule opération. On utilise pour cela un ***DataAdapter***.

7.2.1. Effectuer les changements

Il existe 2 manières d'actualiser les ***DataRows***. On peut modifier une valeur à la fois :

```
dataRow["FieldName"] = "FieldValue";
```

On peut également changer plusieurs valeurs en une seule opération :

```
dataRow.BeginEdit();
dataRow["Field1"] = "Value1";
```

```
dataRow["Field2"] = "Value2";
dataRow.EndEdit();
```

Un ***DataRow*** contient plusieurs versions des données pour un même champ. Voici les différentes valeurs de l'énumération ***DataRowVersion*** :

- ***Current*** : c'est la valeur accessible en lecture
- ***Original*** : c'est la valeur originale du champ, quand la table a été créée
- ***Proposed*** : c'est la nouvelle valeur après le ***BeginEdit*** mais avant ***EndEdit***
- ***Default*** : c'est la valeur par défaut appliquée aux nouvelles lignes

Par défaut, on accède toujours à la valeur ***Current*** d'un ***DataRow***. Si besoin, on peut accéder aux différentes versions d'un champ, en utilisant :

- ***Row.HasVersion(DataRowVersion.Original)*** pour déterminer si ce type de valeur existe
- ***Row["FieldName", DataRowVersion.Original]*** pour récupérer la valeur correspondante à la version spécifiée

Pour ajouter un ***DataRow***, on utilise la méthode ***Add***. Pour supprimer un ***DataRow*** utiliser la méthode ***Delete***. ***Delete*** positionne ***RowState*** à ***Deleted***, mais laisse le ***DataRow*** dans la collection. Pour réellement supprimer un ***DataRow***, il faut appeler ***Remove***.

Attention : si l'enregistrement est « removed », il ne sera pas pris en compte dans la phase 2 du processus d'update du DataAdapter et l'enregistrement correspondant dans la base ne sera pas supprimé.

Un ***DataRow*** possède également un état spécifié dans sa propriété ***RowState***. Cela permet de parcourir un ***DataTable*** d'un ***DataSet*** et de savoir si un ***DataRow*** est nouveau, a été supprimé, … L'énumération ***RowState*** possède les valeurs suivantes :

- ***Added*** : le ***DataRow*** a été ajouté dans la collection et ***AcceptChanges*** n'a pas été appelé.
- ***Deleted*** : le ***DataRow*** a été supprimé avec la méthode ***Delete***.
- ***Detached*** : le ***DataRow*** a été créé, mais n'a pas encore été attaché à une table du ***DataSet***.
- ***Modified*** : le ***DataRow*** a été modifié et ***AcceptChanges*** n'a pas été appelé.
- ***Unchanged*** : le ***DataRow*** n'a pas subi de modification depuis le dernier appel à ***AcceptChanges*** ou depuis la création du ***DataTable***.

7.2.2. Transférer les changements

Une fois que vous avez modifié vos lignes ou vos valeurs dans un ***DataSet***, il faut appeler la méthode ***Update*** du ***DataAdapter***, en passant en paramètre le ***DataSet***.

```
dataAdapter.Update(dataSet);
```

La méthode ***Update*** du ***DataAdapter*** va parcourir toutes les ***DataTable*** du ***DataSet***, et pour chaque ***DataTable***, va parcourir tous les ***DataRow***. Pour chaque ***DataRow***, il va analyser son état grâce à la propriété ***RowState***, et en fonction de cet état, il va exécuter la ***Command*** appropriée. En effet, le ***DataAdapter*** peut être lié à des objets ***Command*** qui contiennent des requêtes UPDATE, INSERT ou DELETE.

Après la mise à jour des données, le ***DataAdapter*** appelle la méthode ***AcceptChanges*** sur le ***DataSet***, ce qui permet d'accepter tous les changements en une seule opération. En clair, la version ***Original*** de chaque ***DataRow*** va prendre la même valeur que celle de la version ***Current***. Ainsi, si une nouvelle mise à jour à lieu par un appel à la méthode ***Update***, aucune mise à jour ne serait effectuée dans la base de données car les versions ***Current*** et ***Original*** sont identiques.
Si besoin dans votre code, vous pouvez appeler la méthode ***AcceptChanges*** ou ***RejectChanges***.

8. DataSet typé

Quand on utilise un ***DataAdapter***, on doit saisir une requête SQL de type SELECT pour récupérer des données. Eventuellement, on peut générer les requêtes de mise à jour.
Une autre fonctionnalité du ***DataAdapter*** est de pouvoir générer un ***DataSet*** typé. En fonction des champs retournés par le SELECT, le ***DataAdapter*** va pouvoir générer une classe dérivée de la classe ***DataSet***, et qui contiendra comme propriétés les champs retournés par le SELECT, et ces propriétés seront du bon type (par exemple, le champ ID sera de type ***Integer***, le champ LastName sera du type string, …). De cette manière, on peut manipuler les bons types de données au niveau de l'application (d'où le terme DataSet typé).

L'autre avantage de cette technique est de pouvoir poser ce ***DataSet*** typé directement sur votre fiche en conception. Cela signifie que vous allez pouvoir poser des composants d'affichage en conception (***TextBox***, ***Label***, ***DataGrid***, …) et les relier (c'est-à-dire effectuer un DataBind) directement sur les bons champs, et cela en conception.

Pour créer un ***DataSet*** Typé, nous avons déjà vu une méthode dans la partie ***DataAdapter*** de ce cours. Une autre méthode est d'utiliser un Schéma XSD. Ce dernier peut être fourni lorsque vous travaillé avec un Web Service mais vous pouvez également le créer vous-même. Nous ne détaillerons pas d'avantage cette dernière technique.

Une fois que votre ***DataSet*** typé est créé, vous pouvez remarquer que celui-ci est très différent par rapport un ***DataSet*** non typé notamment par l'apparition de propriétés dédiées aux données contenues.
Citons par exemple l'apparition de propriétés du nom des tables présentes dans le ***DataSet***. Elle permet de vous donner plus facilement accès aux données des tables comme, par exemple, avec des propriétés représentants les colonnes (« nomcolonneColumn »).