Chapitre 7 : accès aux données

7.1 Connecteur ADO.NET

7.1.1 Généralités

La nouveauté de l'ADO.NET par rapport à son ancêtre l'ADO est la gestion de données dans une application dans un environnement déconnecté. Ce mode, par rapport au mode connecté classique, possède plusieurs avantages et inconvénients.

7.1.2 Les fournisseurs de données

Chaque fournisseur est relié à une base de données propre, c'est-à-dire qu'il est compatible à l'API de sa base de données. Cependant, les bases de données peuvent implémenter plusieurs API (par exemple en installant certains pilotes comme ODBC pour l'ODBC) :

Fournisseur	Description
SQL Server	Les classes de ce fournisseur se trouvent dans l'espace de nom *System.Data.SqlClient*, chaque nom de ces classes est préfixé par *Sql*. SQL Server à accès au serveur sans utiliser d'autres couches logicielles le rendant plus performant.
OLE DB	Les classes de ce fournisseur se trouvent dans l'espace de nom *System.Data.OleDb*, chaque nom de ces classes est préfixé par *OleDb*. Ce fournisseur exige l'installation de MDAC (Microsoft Data Access Components). L'avantage de ce fournisseur est qu'il peut dialoguer avec n'importe quelle base de données le temps que le pilote OLE DB est installé dessus, mais par rapport à SQL server, OLE DB utilise une couche logicielle nommée OLE DB ; il requiert donc plus de ressources diminuant par conséquent les performances.
ODBC	Les classes de ce fournisseur se trouvent dans l'espace de nom *System.Data.Odbc*, chaque nom de ces classes est préfixé par *Odbc*. Tout comme l'OLE DB, ODBC exige l'installation de MDAC. Il fonctionne avec le même principe qu'OLE DB mais au lieu d'utiliser une couche logicielle, il utilise le pilote ODBC.
Oracle	Les classes de ce fournisseur se trouvent dans l'espace de nom *System.Data.OracleClient*, chaque nom de ces classes est préfixé par *Oracle*. Il permet simplement de se connecter à une source de données Oracle.

Remarque :

SQL Server et Oracle sont tout deux des fournisseurs de données managés. C'est-à-dire qu'ils sont optimisés pour certains types de bases de données.

7.1.3 Accéder à la base de données

Pour dialoguer avec la base de données, tous ces fournisseurs implémentent six classes de bases :

Classe	Description
Command	Stocke les informations sur la commande et permet son exécution sur le serveur de base de données.
CommandBuilder	Permet de générer automatiquement des commandes ainsi que des paramètres pour un *DataAdapter*.
Connection	Permet d'établir une connexion à une source de données spécifiée.
DataAdapter	Permet le transfert de données de la base de données vers l'application et inversement (par exemple pour une mise à jour, suppression ou modification de données). Il est utilisé en mode déconnecté (voir partie 5.4).
DataReader	Permet un accès en lecture seule à une source de données.
Transaction	Représente une transaction dans le serveur de la base de données.

Remarque :

La classe Connection n'apparaît pas dans le Framework 3.5. En effet, les classes des fournisseurs managés ont leur propre classe tel que SqlConnection.

7.1.4 Interface portable

Les classes de chaque fournisseur varient, et donc par conséquence, le code devient spécifique à un fournisseur. Mais il existe une solution pour remédier à ce problème : on peut utiliser comme type de données les interfaces qu'elles implémentent.

En effet, les classes spécifiques aux fournisseurs permettront juste d'établir la connexion, on pourra ensuite utiliser que ces interfaces. Les six classes données dans le tableau précédent implémentent leurs interfaces respectives :

Interface	Description
IDataAdapter	Permet de remplir et actualiser un objet *DataSet* et de mettre à jour une source de données.
IDataReader	Permet de lire un ou plusieurs flux de données en lecture seule à la suite de l'exécution d'une commande.
IDataParameter	Permet d'implémenter un paramètre pour une commande.
IDbCommand	Permet de donner une commande qui s'exécutera au moment de la connexion à une source de données.
IDbConnection	Représente une connexion unique avec une source de données.
IDbDataAdapter	Représente un jeu de méthodes qui permet d'exécuter des opérations sur des bases de données relationnelles (insertion, sélection, …).
IDbTransaction	Représente une transaction à exécuter au niveau d'une source de données.

Ces interfaces ne montrent pas l'étendue des possibilités que peut donner un fournisseur managé. En effet, les fournisseurs managés comprennent leurs propres classes permettant plus d'actions ou en les améliorant.
Voici un exemple de code utilisant ces interfaces :

```
using System.Data;
using System.Data.Common;
using System.Data.SqlClient;

namespace TesterInterface
{
    class Program
    {
        static void Main(string[] args)
        {
            string requete, connexionString;
            connexionString = @"Data Source=.\\SQLEXPRESS;Initial
Catalog=Emp;Integrated Security=true;";
            requete = "SELECT * FROM Employe"; IDbConnection connexion =
new SqlConnection(connexionString);
            IDbCommand commande = connexion.CreateCommand();
            commande.CommandText = requete;
            commande.CommandType = CommandType.Text;
            connexion.Open();
            IDataReader lire = commande.ExecuteReader();
            while (lire.Read())
            {
                Console.WriteLine("ID : {0} | Nom : {1} | Prenom : {2} |
RoleNumero : {3}", lire.GetInt32(0), lire.GetString(1), lire["Prenom"],
lire["Role"]);
            }
            connexion.Close();
            connexion.Dispose();
            Console.ReadLine();
        }
    }
}
```

7.1.5 Mode connecté / Mode déconnecté

L'ADO.NET permet de séparer les actions d'accès ou de modification d'une base de données. En effet, il est possible de manipuler une base de données sans être connecté à celle-ci, il suffit juste de se connecter pendant un court laps de temps afin de faire une mise à jour. Ceci est possible grâce au DataSet. C'est pourquoi, il existe deux types de fonctionnements :

- Le mode connecté
- Le mode déconnecté

Ci-après, la différence par avantages et inconvénients :

Mode	Avantages	Inconvénients
Connecté	Avec un mode connecté, la connexion est permanente, par conséquence les données sont toujours à jour. De plus il est facile de voir quels sont les utilisateurs connectés et sur quoi ils travaillent. Enfin, la gestion est simple, il y a connexion au début de l'application puis déconnexion à la fin.	L'inconvénient se trouve surtout au niveau des ressources. En effet, tous les utilisateurs ont une connexion permanente avec le serveur. Même si l'utilisateur n'y fait rien la connexion gaspille beaucoup de ressource entrainant aussi des problèmes d'accès au réseau.

Déconnecté	L'avantage est qu'il est possible de brancher un nombre important d'utilisateurs sur le même serveur. En effet, ils se connectent le moins souvent et durant la plus courte durée possible. De plus, avec cet environnement déconnecté, l'application gagne en performance par la disponibilité des ressources pour les connexions.	Les données ne sont pas toujours à jour, ce qui peut aussi entrainer des conflits lors des mises à jour. Il faut aussi penser à prévoir du code pour savoir ce que va faire l'utilisateur en cas de conflits.

Il n'existe pas un mode meilleur que l'autre, tout dépend de l'utilisation que l'on compte en faire.

7.2 Établir une connexion

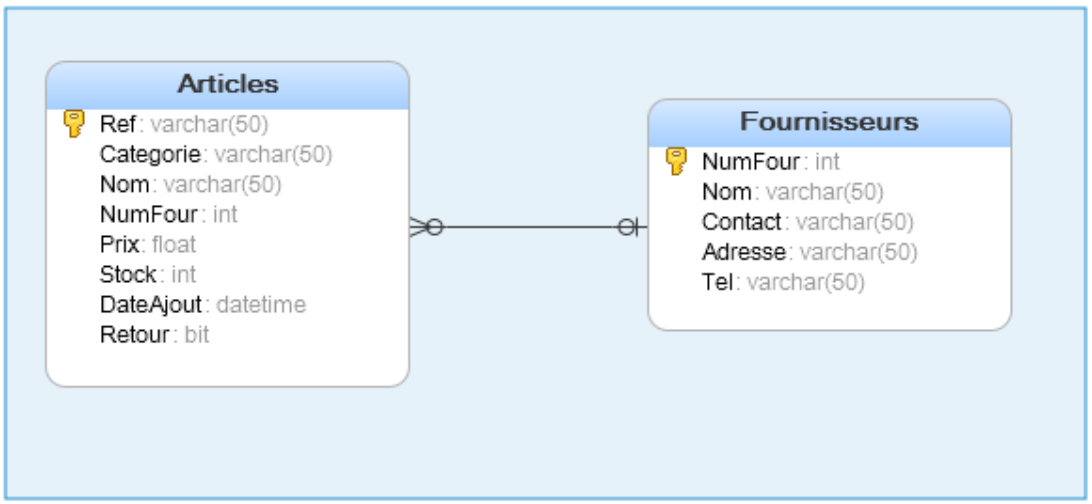
7.2.1 Les chaînes de connexions

Dans un mode connecté, il faut tout d'abord connecter l'application à la base de données. Nous utiliserons SQL Server 2005 pour la suite. Pour ouvrir cette connexion il faut d'abord déclarer une variable, ici ce sera « connexion » :

```
SqlConnection connexion
```

Soit la base de données SQLServer nommée 'BDStock.mdb', une table de cette base est la table 'Articles', l'autre est la table Fournisseurs

Vue de la base de données BDStock



La propriété *ConnectionString* permet d'indiquer les paramètres de connexion. Cela se fait sous forme de chaîne de caractères, tel que par exemple :

```
connexionString = @"Data Source=.\\SQLEXPRESS; Initial
Catalog=BDStock;Integrated Security=True" ;
```

Voici les différents paramètres disponibles dans une *ConnectionString* :

Paramètre	Description
Connect Timeout	Indique le temps d'attente de connexion en seconde. Ce laps de temps dépassé, une exception est levée.
Connection LifeTime	Indique la durée de vie d'une connexion dans un pool, la valeur 0 (zéro) correspond à l'infini.
Connection Reset	Indique si la connexion a été réinitialisée lors de son retour dans un pool.
Data Source	Indique le nom ou l'adresse réseau du serveur.
Initial Catalog	Indique le nom de la base de données où l'application doit se connecter.
Integrated Security	Indique s'il faut un nom et un mot de passe. Si la valeur et sur False, un login et password seront demandés.
Max Pool Size	Indique le nombre maximum de connexion dans un pool. Par défaut, le nombre maximum de connexions est 100.
Min Pool Size	Indique le nombre minimum de connexion dans un pool.
Persist Security Info	Indique si le nom et le mot de passe est visible par la connexion.
Pwd	Indique le mot de passe associé au compte SQL Server.
Pooling	Indique si une connexion peut être sortie d'un pool.
User ID	Indique le nom du compte SQL Server.

Afin de vérifier l'état d'une connexion, ADO.NET propose l'énumération *ConnectionState*. Il possède différentes propriétés :

Propriété	Description
Broken	Permet de savoir si la connexion est interrompue, cette connexion peux se fermer puis se rouvrir.
Closed	Permet de savoir si l'objet connexion est fermé.
Connecting	Permet de savoir si l'objet connexion est en cours de connexion.
Executing	Permet de savoir si une commande est en train de s'exécuter.
Fetching	Permet de savoir si l'objet connexion est en train de récupérer des données.
Open	Permet de savoir si l'objet connexion est ouvert.

7.2.2 Les pools de connexions

Afin de réduire le coût en ressource engendré par les connexions à des bases de données, l'ADO.NET propose une technique d'optimisation : le pool de connexion. Lorsque qu'une application ouvre une nouvelle connexion, un pool est créé. Les pools permettent de stocker toutes les requêtes récurrentes.

Chaque fois qu'un utilisateur ouvre une connexion avec la même ConnectionString qu'un pool, le dispositif de connexion vérifie s'il y a une place disponible dans ce pool, si le MaxPoolSize n'est pas atteint, la connexion rentre dans l'ensemble. Un pool est effacé lorsqu'une erreur critique est levée.

Les pools sont paramétrables dans le ConnectionString et une connexion est retirée d'un pool lorsqu'elle est inactive depuis une certaine durée.

Voici les mots-clés de connexion de votre pool de connexion :

Nom	Par défaut	Description
Connection Lifetime	0	Quand une connexion tente de rejoindre le pool, si son temps de connexion dure plus de *x* secondes (*x* étant la valeur de la propriété), la connexion est stoppée.
Connection Reset	*True*	Détermine si la connexion est remise à zéro lors de la création d'un ensemble de connexion.
Enlist	*True*	Si vous utilisez une connexion dans le cadre d'une transaction, vous pouvez définir ce mot-clé sur *True*.
Load Balance Timeout	0	Indique le nombre de secondes d'une connexion avant quelle soit détruite de l'ensemble.
Max Pool Size	100	Indique le nombre maximum de connexions autorisées dans un ensemble pour une chaîne de connexion spécifique. En d'autres termes si votre connexion demande sans cesse de se connecter à la base de données, vous pourriez avoir besoin d'augmenter votre *Max Pool Size* (par défaut 100 connexions autorisés).
Min Pool Size	0	Détermine le nombre minimum de connexions autorisés.
Pooling	True	Indique une valeur booléenne si la connexion est regroupée (True) ou si elle est ouverte à chaque demande de connexion.

En plus des mots-clés permettant de contrôler le comportement des *Connection Pool*, il existe des méthodes qui ont des incidences sur un ensemble :

Nom	Object	Description
ClearAllPool	SqlConnection et OracleConnection	Réinitialise toutes les *Connection Pool*.
ClearPool	SqlConnection et OracleConnection	Réinitialise une *Connection Pool* spécifique.
ReleaseObjectPool	OleDbConnection et Odbcconnection	Spécifie au pool de connexions qu'il peut être détruit lorsque la dernière connexion qu'il contenait a été détruite.

7.2.3 Déconnexion

Pour couper la connexion entre l'application et la base de données, il suffit d'écrire :
`nomConnexion.Close();`

7.3 Mode connecté

7.3.1 Les commandes

Contrairement à une base de données, les requêtes SQL et les procédures stockées sont exécutées à partir de commandes. Les commandes contiennent toutes les informations nécessaires à leur exécution et effectuent des opérations telles que créer, modifier ou encore supprimer des données d'une base de données.

Vous utilisez ainsi des commandes pour faire des exécutions de requêtes SQL qui renvoient les données nécessaires. Remarque : les requêtes SQL et les procédures stockées sont deux choses différentes. En effet les procédures stockées sont des requêtes SQL déjà enregistrées dans la mémoire cache du serveur. Chaque fournisseur de base de données possède leurs propres objets Command qui sont les suivantes :

Nom	Type de sources de données
SqlCommand	SQL Server
OleDbCommand	OLE DB
OdbcCommand	ODBC
OracleCommand	Oracle

Il existe plusieurs propriétés et méthodes communes à chaque fournisseur pour gérer des commandes, voici les principales :

Propriétés	
Nom	**Description**
CommandText	Permet de définir l'instruction de requêtes SQL ou de procédures stockées à exécuter. Lié à la propriété *CommandType*.
CommandTimeout	Permet d'indiquer le temps en secondes avant de mettre fin à l'exécution de la commande.
CommandType	Permet d'indiquer ou de spécifier la manière dont la propriété CommandText doit être exécutée.
Connection	Permet d'établit une connexion.
Parameters	C'est la collection des paramètres de commandes. Lors de l'exécution de requêtes paramétrées ou de procédures stockées, vous devez ajouter les paramètres objet dans la collection.
Transaction	Permet de définir la SqlTransaction dans laquelle la SqlCommand s'exécute.
Cancel	Permet de tenter l'annulation de l'exécution d'une commande.
ExecuteNonQuery	Permet d'exécuter des requêtes ou des procédures stockées qui ne retournent pas de valeurs.
ExecuteReader	Permet d'exécuter des commandes et les retourne sous forme de tableau de données (ou des lignes).
ExecuteScalar	Permet d'exécuter les requêtes ou les procédures stockées en retournant une valeur unique.

ExecuteXMLReader	Permet de retourner les données sous le format XML.

Vous pouvez aussi manipuler des événements. Voici les deux principaux :

Evènements	Description
Disposed	Permet d'appeler la dernière méthode avant que l'objet ne soit détruit.
StatementCompleted (seulement pour SqlCommand)	Se produit lorsque l'exécution d'une instruction se termine.

7.3.2 Utiliser des commandes

Une fois la connexion établie, la classe *SqlCommand* permet d'appeler la méthode *CreateCommand* qui permettra l'exécution de commandes SQL. Il existe trois méthodes afin de créer une commande :

- Vous pouvez directement utiliser un des constructeurs de la classe *SqlCommand*. Par contre cela nécessite l'utilisation de deux propriétés : *CommandText* et *Connection*. Voici un exemple utilisant cette méthode :

```
SqlCommand commande= new SqlCommand();
commande.Connection = connexion;
commande.CommandText = "SELECT * FROM Articles ";
```

- La deuxième méthode est l'utilisation d'un constructeur surchargé, voici par exemple :

```
commande = new SqlCommand("SELECT * FROM Articles", connexion);
```

- La dernière méthode est d'utiliser la méthode CreateCommand de l'objet de connexion comme dans l'exemple ci-dessous :

```
SqlCommand commande = connexion.CreateCommand();
commande.CommandText = "SELECT * FROM Articles";
```

Le fonctionnement pour exécuter une procédure stockée est quasiment identique à l'exécution d'une requête SQL classique. Il faut que la propriété *CommandText* contienne le nom de la procédure, par contre la propriété *CommandType* doit prendre la valeur *StoredProcedure* au lieu de *Text*.

L'avantage d'une procédure stockée est une amélioration de la performance car la procédure se trouve précompilée dans le cache du serveur.

Voici un exemple qui permet d'afficher toutes les informations d'un utilisateur de la table Articles:

```
using System;
using System.Data;
using System.Data.Common;
using System.Data.SqlClient;

namespace TesterInterface
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            SqlConnection connexion = new SqlConnection();
            connexion.ConnectionString = "Data Source=.\\sqlexpress;Initial
            Catalog=BDStock;Integrated Security=True";
            SqlCommand command = connexion.CreateCommand();
            connexion.Open();

            string id;
            string requete = "RecupInformation";
            command.CommandText = requete;
            command.CommandType = CommandType.StoredProcedure;

            Console.WriteLine("Quel est la référence de l'article sur
            laquelle vous voulez les informations ?");
            id = Console.ReadLine();
            SqlParameter paramIdToKnow = new SqlParameter("@IdToKnow", id);

            command.Parameters.Add(paramIdToKnow);

            IDataReader lecture = command.ExecuteReader();

            while (lecture.Read())
            {
                  Console.WriteLine("Id : {0} Nom : {1} Prenom : {2} Role :
                  {3}", lecture["Ref"], lecture.GetString(1),
                  lecture.GetString(2), lecture.GetInt32(3));
            }
            connexion.Close();
            connexion.Dispose();
            Console.ReadLine();
        }
    }
}
```

Remarquons que le champ récupéré est typé (ici le type string grâce à GeString);
On distingue d'autres types:
GetDateTime, GetDouble, GetGuid, GetInt32, GetBoolean, GetChar, GetFloat, GetByte, GetDecimal etc.;

Il est possible de récupérer des données sans typage: on aurait écrit ***myReader(0)*** ou utiliser **GetValue** (on récupère un objet).

Read() avance la lecture des données à l'enregistrement suivant , il retourne True s'il y a encore des enregistrements et False quand il est en fin de la table; cela est utilisé pour sortir de la boucle Do Loop.

Afin d'exécuter une instruction SQL qui renvoie plusieurs valeurs, on peut utiliser la méthode *ExecuteReader*; elle va retourner l'objet *Datareader* qui va permettre la lecture des données. Si l'instruction SQL ne doit renvoyer qu'une valeur unique, on peut utiliser la méthode *ExecuteScalar* qui va à la fois s'occuper de l'exécution et retourner la valeur.

Vous pouvez également exécuter des commandes qui vous renvoient les données au format XML. Pour cela vous devez régler la propriété *CommandText* de votre instruction SQL au format XML puis faire appel à la méthode *ExecuteXMLReader* qui retourne un objet *XmlReader* (dont la classe est stockée dans *System.Xml*).

Lorsque vous configurez votre commande, vous pouvez utiliser le *Query Designer* afin de créer et de personnaliser vos requêtes. Donc vous devez sélectionner votre base de données dans le *Server Explorer* puis cliquer sur *New Query*. Vous ouvrez ainsi le *Query Designer* et pourrez alors non seulement créer votre requête mais en plus la personnaliser (tableau, affichage…).

Durant un échange entre une application et une base de données, l'application est bloquée durant l'attente de la réponse du serveur. Pour remédier à ce problème, l'ADO.NET propose les commandes asynchrones. En effet, ces commandes permettent à l'application de faire autre chose en attendant la réponse du serveur. Voici les méthodes qui sont utilisées lors d'un processus asynchrone :

Méthodes	Description
BeginExecuteNonQuery	Commence la version asynchrone de la méthode *ExecuteNonQuery*.
BeginExecuteReader	Commence la version asynchrone de la méthode *ExecuteReader*.
BeginExecuteXmlReader	Commence la version asynchrone de la méthode *ExecuteXmlReader*.
EndExecuteNonQuery	Appeler cette méthode après l'événement *StatementCompleted* afin d'achever l'exécution de la commande.
EndExecuteReader	Appeler cette méthode après l'événement *StatementCompleted* afin de renvoyer le *DataReader* avec les données retournées par la commande.
EndExecuteXmlReader	Appeler cette méthode après l'événement *StatementCompleted* afin de renvoyer le *XmlReader* avec les données retournées par la commande.

7.3.3 Les paramètres de commandes SQL

Un paramètre peut être considéré comme un type de variable qui permet de transmettre des valeurs et des retours entre votre demande et la base de données. Comme toute variable dans une application, les paramètres sont créés pour contenir un certain type de données. Les types de données des paramètres sont assignés en utilisant les types définis dans l'énumération de l'objet *System.Data.SqlDbType*. Cette énumération contient toute une liste des types disponibles dans SQL Server.

Vous définissez un paramètre à une requêtes SQL (ou à une procédure stockée) lorsque vous changez les critères de votre requêtes rapidement. Par exemple l'utilisation typique d'utilisation d'un paramètre est dans la clause WHERE de votre requête SQL. Les paramètres vous permettent aussi de contrôler la façon dont est entré un utilisateur dans une requête. Remarque : Pour SQL Server le symbole @ est utilisé pour créer des paramètres nommés. Le symbole point d'interrogation ? (paramètre anonyme) est utilisé dans les autres types de base de données.

7.3.4 Les types de paramètres

La modification des informations contenue dans votre base de données est faite par les instructions SQL. Il existe quatre types de paramètres :
- le premier est de type *Input*, c'est-à-dire que vous voulez utiliser un paramètre pour envoyer des données à la base de données.

- le second est le type *Output* qui lui est utilisé lorsqu'on veut récupérer des données.

- le troisième est *l'InputOutput* qui est exploité pour faire les deux actions précédentes, c'est-à-dire envoyer et récupérer des données.

- enfin le dernier type est le *ReturnValue* qui retourne simplement une valeur assignée.

7.3.5 Créer un paramètre

Paramétrer vos requêtes sert aussi à les rendre génériques. Les paramètres servent à prendre un emplacement dans une requête qui sera plus tard utilisé dans votre code.

La classe *SqlParameter* permet de créer des objets de type *Sqlparameter* contenant : le nom, la valeur et la direction d'utilisation du paramètre. Voir TP9

7.3.6 Les BLOBs

Les *BLOBs* dans une base de données ne sont pas de simples données de types chaines de caractères, ce sont des les types de données binaires du type graphiques, photos, documents enregistrés en format binaire ou bien des exécutables (ils peuvent contenir tous les types), par conséquence leur utilisation est plus complexe.

La taille d'un *BLOB* peut dépasser plusieurs Go et par conséquence peut nuire aux performances au moment d'un chargement. En revanche le .NET Framework fournit des classes permettant le déplacement de grosses quantités de données binaires. Ces classes (comme *BinaryReader* ou *BinaryWriter*) se trouvent dans l'espace de nom *System.IO*.

7.3.7 Copier un grand nombre de données

Pour copier un grand nombre de données vers une table de données de façon performante (c'est-à-dire sans trop utiliser de ressources et de temps) il existe deux applications :

- Le Framework .NET qui propose dans le namespace System.Data.SqlClient l'objet SqlBulkCopy.

- SQL Server qui propose la requête BULK INSERT SQL.

Ces solutions permettent dans la majorité des cas de rendre plus performant le transfert.

7.3.8 Les transactions

Les transactions permettent de regrouper des commandes SQL dans une même entité. La transaction permettra que si une des commandes échoue alors l'opération sera arrêtée et la base de données retrouvera son état initial. Pour créer une transaction, il suffit d'instancier votre *Transaction* puis de l'assigner en appelant la méthode *BeginTransaction* à la connexion. Voici un exemple de création d'une transaction :

```
'Création d'une transaction
SqlTransaction transaction
'Définit la transaction à votre connexion
transaction = VotreConnexion.BeginTransaction() ;
```

Les transactions sont managées au niveau de la connexion. Par conséquent nous pourrons commencer une transaction en ouvrant une connexion avec une base de données

pour ensuite commencer les transactions en appelant la méthode *BeginTransaction* issues d'une instance de la classe *SqlTransaction*. Puis vous devez définir quelle commande nécessite une transaction. Enfin à la fin du traitement des données vous avez la possibilité soit de valider vos transactions grâce à la méthode *Commit* soit de les annuler grâce à la méthode *Rollback*.

Exercice

Ecrire le code C# permettant de copier les nom des différents articles dans une liste bix de nom ListBox1

Correction

```
using System.Data.SqlClient
SqlConnection MyConnexion = new SqlConnection("Data
Source=.\\SQLEXPRESS; Initial Catalog=BDStock;Integrated
Security=True") ;
SqlCommand Mycommand = MyConnexion.CreateCommand();
Mycommand.CommandText = "SELECT * FROM Articles";
SqlDataReader myReader = Mycommand.ExecuteReader();
MyConnexion.Open() ;
OleDbDataReader myReader = Mycommand.ExecuteReader() ;
While (myReader.Read())
  ListBox1.Items.Add(myReader.GetString(2)) ;
myReader.Close() ;
MyConnexion.Close() ;
```

Remarque :

Avec **ExecuteScalar** de l'objet **Command** on peut récupérer les résultats d'une requête Sql qui contient une instruction **COUNT** (comptage) **AVG** (moyenne) **MIN** (valeur minimum) **MAX** (valeur maximum) **SUM** (somme)

Exemple :

Compter le nombre d'articles:

```
 Mycommand.CommandText = "SELECT COUNT(*) FROM Articles";
MyConnexion.Open();
int Resultat = (int)Mycommand.ExecuteScalar();
```

- Chaque SGDB a des erreurs spécifiques. Pour les détecter il faut utiliser les types d'erreur spécifiques: **SqlException** et **OleDbException** par exemple:

Exemple d'interception d'erreur:

```
try
{
    MyConnexion.Open();
}
catch (OleDbException ex)
{
   MessageBox.show(ex.Message);
}
```

7.4 Utilisation du DataSet

7.4.1 Généralités

Le **DataSet** est une représentation en mémoire des données. On charge le DataSet à partir de la base de données. Une fois chargé on peut travailler en mode déconnecté. Pour effectuer une modification, on modifie le DataSet puis on met à jour la base de données à partir du DataSet.

Pour remplir un DataSet il faut une **Connexion** puis un **DataAdapter** qui par sa propriété Fill charge le DataSet.
.

Les données sont extraites à l'aide de **requête SQL** sur l'objet **Command**, et on utilise un **CommandBluider** pour mettre à jour la base de donnée à partir du DataSet.

Le DataSet est organisé comme une base de données **en mémoire**, il possède:

- Une propriété **Tables** qui contient des **DataTable** (Comme les tables d'un BD)
- Chaque **DataTable** contient une propriété **Columns** qui contient les **DataColomn** (les colonnes ou champs de la BD) et une propriété **Rows** qui contient des **DataRow** (Les lignes ou enregistrements de la BD)

DataColumn contient des informations sur le type du champ.
DataRow permet d'accéder aux données.

Un DataSet possède aussi la propriété **Constraints** qui contient les **Constraint** (Clé primaire ou clé étrangère), et la propriété **Relations** qui contient les **DataRelations** (Relation entre les tables).

Il est possible de créer des **DataTable** 'autonomes' et y mettre une table provenant d'un dataSet:

Il est possible aussi de créer des **DataView** (DataView trié, DataView dont les lignes ont été filtrées (RowFilter)) qui sont des vues, représentatives d'une ou plusieurs tables.

Une DataTable ou un DataView peut être affichés dans un contrôle (**DataGridView** par exemple).

Ici on connecte la base de donnée au DataSet par l'intermédiaire de Connexion et **DataAdapter**. On peut soit :

- utiliser les membres du DataSet pour lire ou modifier les données.
- lier le DataSet, ou le DataTable ou le DataView à un contrôle DataGrid ou ListBox, pour prévoir un affichage automatique.

7.4.2 Utilisation du DataSet et du DataView à travers un exemple

Soit une **base de données SSLServer 'BDStock'**. On veut se connecter à cette base, extraire les enregistrements de la table 'Articles' et les mettre dans un DataSet. Chaque ligne du DataSet contient un Article.

- *importer l'espace de nom permettant d'utiliser les DataSet et OleDB.*

```
using System.Data
using System.Data.SQLClient
```

- *déclarer les objets Ado:*

```
// Déclaration Objet Connexion
private  OleDbConnection ObjetConnection  ;
// Déclaration Objet Commande
private  OleDbCommand ObjetCommand ;
// Déclaration Objet DataAdapter
private OleDbDataAdapter ObjetDataAdapter ;
// Déclaration Objet DataSet
private DataSet ObjetDataSet = new DataSet() ;
// String contenant la 'Requête SQL'
private String strSql;
// Déclaration Objet DataTable
private DataTable ObjetDataTable;
// Déclaration Objet DataRow (ligne)
private DataRow ObjetDataRow;
//Numéro de la ligne en courante ou de l'enregistrement courant
private int  RowNumber;
//Paramêtres de connexion à la DB
private String strConn;
//Pour recompiler les données modifiées avant de les remettre dans le
"DataAdapter"
private OleDbCommandBuilder ObjetCommandBuilder ;
```

- *Ouverture*

```
//Initialisation de la chaîne de paramètres pour la connexion
strConn = " Data Source=.\\sqlexpress;Initial Catalog=BDStock;Integrated
Security=True ";
//Initialisation de la chaîne contenant l'instruction SQL
strSql = "SELECT Articles.* FROM Articles";
//Instanciation d'un Objet Connexion
ObjetConnection = new OleDbConnection();
//Donner à la propriété ConnectionString les paramètres de connexion
ObjetConnection.ConnectionString = strConn;
//Ouvrir la connexion
ObjetConnection.Open();
//Instancier un objet Commande
ObjetCommand = new OleDbCommand(strSql);
//Instancier un objet Adapter
ObjetDataAdapter = new OleDbDataAdapter(ObjetCommand);
//initialiser l'objet Command
ObjetCommand.Connection() = ObjetConnection;
//Avec l'aide de la propriété Fill du DataAdapter charger le //DataSet
ObjetDataAdapter.Fill(ObjetDataSet, "Articles");
//Mettre dans un Objet DataTable une table du DataSet
ObjetDataTable = ObjetDataSet.Tables["Articles"];
```

Ici le code est simplifié, mais en réalité, il est indispensable de travailler avec Try Catch pour capter les exceptions surtout pour ObjetConnection.Open() et pour l'update.

- *Affichage d'un enregistrement:*

Soient un TextBox nommée 'TXTNom' et un TextBox 'TXTRef' qui afficheront les noms et les références des articles.

On rappelle que **RowNumber** est une variable contenant le numéro de la ligne en cours (allant de **0** à **Rows.Count-1**)

```
if (RowNumber < 0)
    exit  ;
//Lors de l'ouverture de la BD, s'il n'y a aucun //enregistrement
if (RowNumber > ObjetDataTable.Rows.Count - 1)
    exit;
// ObjetTable.Rows(Numéro de lignes).Item( Nom de colonne) donne le contenu
d'un champ dans une ligne donnée
TXTNom.Text = (ObjetDataTable.Rows[RowNumber]["Nom"]).ToString();
TXTRef.Text = (ObjetDataTable.Rows[RowNumber]["Ref"]).ToString();
//Item peut avoir en paramètre le nom de la colonne ou sont //index
//On suppose que les deux zones texte TXTNon et TXTRef
//existent déjà.
```

- *Pour se déplacer*

Voir la ligne suivante par exemple:

```
RowNumber += 1
//incrémente le numéro de la ligne en cours
RowNumber -= 1 // pour la précédente.
RowNumber = 0 // pour la première.
RowNumber = ObjetDataTable.Rows.Count-1; //pour la dernière.
```

- *Modifier un enregistrement*

```
//Extraire l'enregistrement courant
ObjetDataRow = ObjetDataSet.Tables["Articles"].Rows[RowNumber];

//Modifier les valeurs des champs en  récupérant le contenu des TextBox
ObjetDataRow["Nom"] = TXTNom.Text;
ObjetDataRow["Ref"] = TXTRef.Text;

//Pour modifier les valeurs changées dans le DataAdapter
ObjetCommandBuilder = new OleDbCommandBuilder(ObjetDataAdapter);

//Mettre à jour
ObjetDataAdapter.Update(ObjetDataSet, "Articles");

//Vider le DataSet et le 'recharger' de nouveau.
ObjetDataSet.Clear();
ObjetDataAdapter.Fill(ObjetDataSet, "Articles");
ObjetDataTable = ObjetDataSet.Tables["Articles"];
```

Remarque :

Quand la commande Update est effectuée, si l'enregistrement ne répond pas à la spécification de la base (doublon, pas de valeur pour une clé primaire, Champ ayant la valeur Null), une exception est levée; si on ne la prévoie pas cela engendre une erreur. Il faut donc mettre la ligne contenant la commande Update dans un try catch.

- *Ajouter un enregistrement*

```
ObjetDataRow = ObjetDataSet.Tables("ARTICLES").NewRow();
ObjetDataRow["Nom"] = Me.TXTnom.Text;
```

```
ObjetDataRow["Ref"] = Me.TXTref.Text;
ObjetDataSet.Tables["ARTICLES"].Rows.Add(ObjetDataRow);
//Pour modifier les valeurs changées dans le DataAdapter
ObjetCommandBuilder = New OleDbCommandBuilder(ObjetDataAdapter);
//Mise à jour
ObjetDataAdapter.Update(ObjetDataSet, "ARTICLES");
//On vide le DataSet et on le 'recharge' de nouveau.
ObjetDataSet.Clear();
ObjetDataAdapter.Fill(ObjetDataSet, "ARTICLES");
ObjetDataTable = ObjetDataSet.Tables["ARTICLES"];
```

- *Effacer l'enregistrement en cours*

```
ObjetDataSet.Tables["ARTICLES"].Rows[RowNumber].Delete() ;
ObjetCommandBuilder = New OleDbCommandBuilder(objetDataAdapter) ;
ObjetDataAdapter.Update(objetDataSet, "ARTICLES") ;
```

- *Fermer la connexion*

```
//Objet connectée
ObjetConnection = Null
ObjetCommand = Null
ObjetDataAdapter = Null

//Objet déconnectée
ObjetDataSet = Null
ObjetDataTable = Null
ObjetDataRow = Null
```

- *Trier, Filtrer, rechercher*

Il existe une méthode Select pour les DataTable mais qui retourne des DataRow.

Exemple :
Lors du chargement du formulaire on ajoute :

```
dtst = new DataSet();

SqlDataAdapter adap = new SqlDataAdapter("select * from
Articles", "Data Source=.\\sqlexpress;Initial
Catalog=BDStock;Integrated Security=True");
adap.Fill(dtst, "Articles");
```

Et le code de la procédure de l'événement TextChanged est le suivant :

```
ObjetDataTable = dtst.Tables[0];
string expression =  "Nom like '" + TXTnom.Text + "%'";
//'expression à rechercher
string sortOrder = "Categorie";
DataRow[] Trouve ;
//résultat dans des DataRow
  Trouve   = ObjetDataTable.Select(expression, sortOrder);
DataTable tab = new DataTable();
tab = ObjetDataTable.Clone();
foreach (DataRow r in Trouve )
  tab.ImportRow(r);

DataView dv= new DataView(tab);
```

```
DataGridView1.DataSource = dv;
```

- *Travailler sur une DataTable.*

Ajouter une ligne (Row)

```
DataRow LeNewRow  = ObjetDataTable.NewRow();
//On crée un DataRow
LeNewRow["NOM"] = "DVD" ;
//On remplit la première cellule

LeNewRow[0] = "128" ;
ObjetDataTable.Rows.Add(LeNewRow) ;
//On ajoute la Row au DataTable
```

- *Utiliser un DataView.*

Il est possible de passer la table dans un DataView et d'utiliser Find, Sort, RowFilter sur le DataView.
Le DataView est un objet qui ressemble à une table mais qui correspond à une représentation et permet les tris ou l'utilisation d'un filtre.

Exemple :

```
//On crée un DataView, on y met une table.
DataView dv ;
dv.Table =  ObjetDataSet.Tables["ARTICLES"];
```

ou

```
dv =new DataView(ObjetDataSet.Tables["ARTICLES"]) ;
```

on peut trier le DataView:

```
dv.Sort = "Nom DESC" ;
```

Pour trier sur le champ Nom en ordre décroissant (ASC pour un tri croissant).

On peut filtrer le DataView:

```
dv.RowFilter = " Categorie = 'Disque'" ;
```

```
// Il faut bien respecter les  ' et les majuscules/minuscules.
```

Le DataView ne contient plus que les rows dont la colonne ="DVD". On peut l'afficher dans une grille.

```
DataGridView1.DataSource = objetDataView;
```

La rechercher dans le DataView est avec Find:

Exemple :

```
dv DataView = new DataView(Mytable);
dv.Sort = "Nom";
```

```
// Cherche les articles de nom "DVD"
      int i = dv.Find("DVD")
```

FindRows retourne des DataRow

```
Object somme = ObjetDataTable.Compute ("Sum (PRIX)", "Categorie='Disque'");
```

On peut aussi utiliser Count() et dans le second paramètre (le filtre))