

ADO.NET

Base de Données

Sommaire

| | |
|---|----|
| ADO.NET Base de Données | 1 |
| 1 Introduction..... | 3 |
| 2 Les bases de données | 3 |
| 2.1 Les fournisseurs de données..... | 3 |
| 2.2 Accéder à la base de données..... | 8 |
| 2.3 Interface portable | 9 |
| 2.4 Mode connecté / Mode déconnecté | 11 |
| 3 Etablir une connexion | 12 |
| 3.1 Les chaînes de connexions..... | 13 |
| 3.2 Les pools de connexions | 15 |
| Spécifie au pool de connexion qu'il peut être détruit lorsque la dernière connexion qu'il contenait a été détruite..... | 18 |
| 3.3 Déconnexion | 18 |
| 4 Mode connecté | 18 |
| 4.1 Les commandes..... | 19 |
| 4.2 Utiliser des commandes..... | 21 |
| 4.3 Les paramètres de commandes SQL..... | 27 |
| 4.4 Les types de paramètres..... | 27 |
| 4.5 Créer un paramètre | 27 |
| 4.6 Les <i>BLOBs</i> | 29 |
| 4.7 Le <i>DataReader</i> | 30 |
| 4.8 Copier un grand nombre de données..... | 30 |
| 4.9 Les transactions | 30 |
| 4.9.1 Niveau d'isolations des transactions | 32 |
| 4.9.2 Les transactions Distribuées | 33 |
| 5 Mode déconnecté | 34 |
| 5.1 Le <i>DataSet</i> | 34 |
| 5.2 Configurer un <i>DataSet</i> | 35 |
| 5.2.1 Créer un <i>DataSet</i> | 35 |

| | | |
|-------|--|----|
| 5.2.2 | Ajouter des <i>DataTable</i> | 35 |
| 5.2.3 | Le <i>DataRelation</i> | 36 |
| 5.2.4 | Fusionner deux <i>DataSet</i> | 37 |
| 5.2.5 | Copier un <i>DataSet</i> | 37 |
| 5.3 | Configurer une <i>DataTable</i> | 38 |
| 5.3.1 | Créer des colonnes contenant des expressions | 38 |
| 5.3.2 | L'incrémentation dans une <i>DataTable</i> | 38 |
| 5.3.3 | Les contraintes dans une <i>DataTable</i> | 38 |
| 5.4 | Le <i>DataAdapter</i> | 39 |
| 5.4.1 | Créer un <i>DataAdapter</i> | 40 |
| 5.4.2 | Créer des commandes pour un <i>DataAdapter</i> | 40 |
| 5.4.3 | Le <i>MissingMappingAction</i> | 40 |
| 5.4.4 | Le <i>MissingSchemaAction</i> | 41 |
| 5.5 | Gérer une <i>DataTable</i> | 42 |
| 5.5.1 | Ajouter des données..... | 42 |
| 5.5.2 | Gérer des données..... | 42 |
| 5.5.3 | Les évènements d'une <i>DataTable</i> | 44 |
| 5.6 | Le <i>DataView</i> | 44 |
| 5.6.1 | Créer un <i>DataView</i> | 45 |
| 5.6.2 | Gérer un <i>DataView</i> | 45 |
| 5.7 | Exemples | 46 |
| 5.7.1 | Remplissage automatique / Lecture de données | 46 |
| 5.7.2 | Remplissage manuel du <i>DataSet</i> | 48 |
| 6 | Conclusion | 50 |

1 Introduction

Ce chapitre porte sur la gestion de base de données grâce à l'ADO.NET. L'ADO.NET permet de relier une application à une base de données. 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. Ces deux modes sont utilisables avec les fournisseurs de base de données qui seront cités plus loin dans le chapitre.

Bon cours .NET

L'équipe ADO.NET.

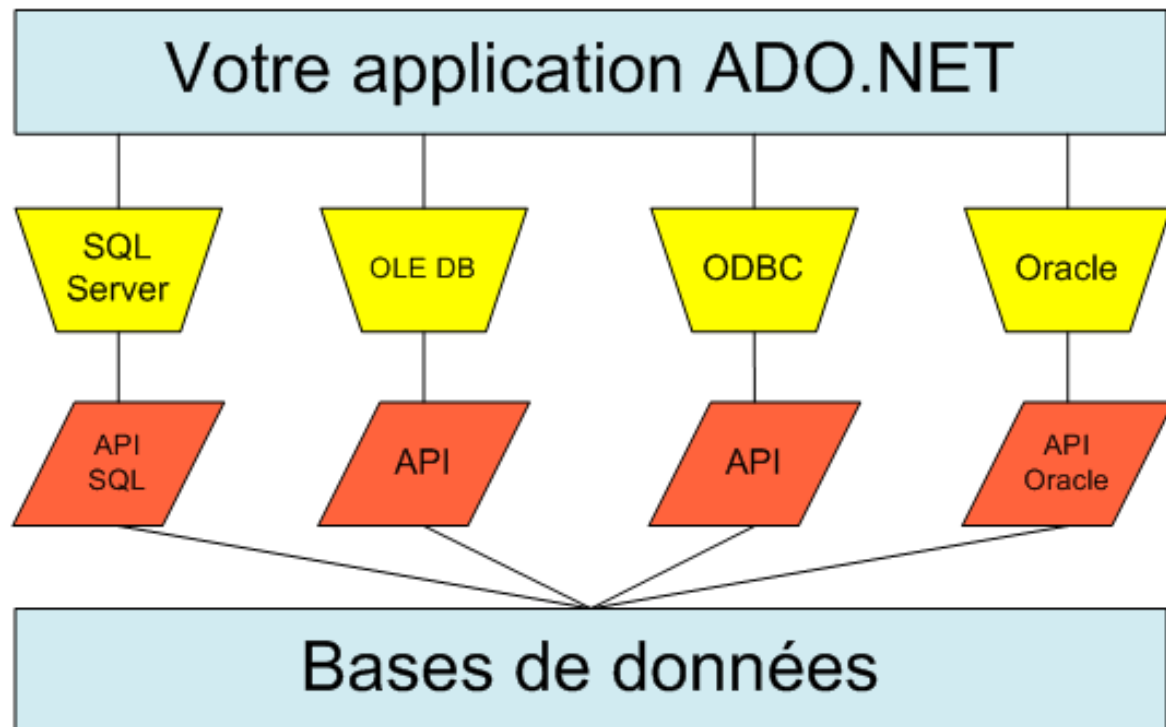
2 Les bases de données

2.1 Les fournisseurs de données

Chaque fournisseur de données permet la communication avec un type de base de données au travers d'une API. Une API (*Application Programming Interface*) est l'interface qui permet l'accès de logiciel par un autre. Ces fournisseurs permettent de récupérer et de transférer des modifications entre l'application et une base de données. Toutes les classes permettant d'utiliser ces fournisseurs se trouvent dans l'espace de nom *System.Data*.

Sur le Framework 3.5, il existe quatre types de fournisseurs :

- Sql Server
- OLE DB
- ODBC
- Oracle



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 <i>System.Data.SqlClient</i> , chaque nom de ces classes est préfixé par <i>Sql</i> . 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 <i>System.Data.OleDb</i> , chaque nom de ces classes est préfixé par <i>OleDb</i> . 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 <i>System.Data.Odbc</i> , chaque nom de ces classes est préfixé par <i>Odbc</i> . 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 <i>System.Data.OracleClient</i> , chaque nom de ces classes est préfixé par <i>Oracle</i> . 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.

Afin que l'accès aux données soit sûr, les fournisseurs de données doivent être disponibles sur le poste de travail. La méthode *GetFactoryClasses* de la classe *DbProviderFactories* permet de donner les fournisseurs disponibles sur le poste de travail en question. Il faut, par contre, ajouter l'espace de nom *System.Data.Common*. Par exemple :

```
'VB
Imports System.Data;
Imports System.Data.Common;

//A rajouter

Sub Main()
    Dim listeFournisseur As DataTable
    listeFournisseur = DbProviderFactories.GetFactoryClasses()

    For Each colonne As DataColumn In listeFournisseur.Columns
        Console.WriteLine(colonne.ColumnName + vbTab)
        ' Affiche le nom des colonnes
    Next
    Console.WriteLine(vbNewLine + vbNewLine)

    For Each ligne As DataRow In listeFournisseur.Rows
        ' Affiche chaque ligne
        For Each colonne As DataColumn In listeFournisseur.Columns
            ' Affiche les cellules
            Console.WriteLine(ligne.Item(colonne.ColumnName) + vbTab)
        Next
        Console.WriteLine(vbNewLine + vbNewLine) ' Retour à la ligne
    Next
End Sub
```

```
//c#

using System.Data;

using System.Data.Common;

//A rajouter

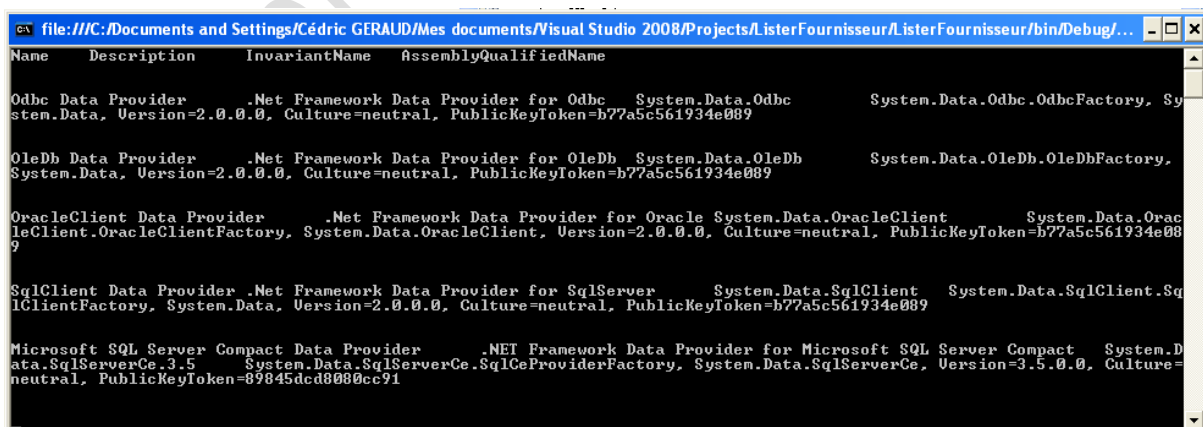
static void Main(string[] args)
{
    DataTable listeFournisseur =
    DbProviderFactories.GetFactoryClasses();

    foreach (DataColumn colonne in listeFournisseur.Columns)
    {
        Console.Write(colonne.ColumnName + "/t");
        //Affiche le nom des colonnes
    }
    Console.WriteLine("/n/n");

    foreach (DataRow ligne in listeFournisseur.Rows)
    //Affiche chaque ligne
    {
        foreach (DataColumn colonne in listeFournisseur.Columns)
        // Affiche les cellules
        {
            Console.Write(ligne[colonne.ColumnName] + "/t");
        }
        Console.WriteLine("/n/n"); //Retour à la ligne
    }

    Console.Read();
}
```

Vous devriez avoir un résultat semblable à :



| Name | Description | InvariantName | AssemblyQualifiedName |
|--|---|--------------------------|---|
| Odbc Data Provider | .Net Framework Data Provider for Odbc | System.Data.Odbc | System.Data.Odbc.OdbcFactory, System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089 |
| OleDb Data Provider | .Net Framework Data Provider for OleDb | System.Data.OleDb | System.Data.OleDb.OleDbFactory, System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089 |
| OracleClient Data Provider | .Net Framework Data Provider for Oracle | System.Data.OracleClient | System.Data.OracleClient.OracleClientFactory, System.Data.OracleClient, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089 |
| SqlClient Data Provider | .Net Framework Data Provider for SqlServer | System.Data.SqlClient | System.Data.SqlClient.SqlClientFactory, System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089 |
| Microsoft SQL Server Compact Data Provider | .NET Framework Data Provider for Microsoft SQL Server Compact | System.Data.SqlServerCe | System.Data.SqlServerCe.SqlCeProviderFactory, System.Data.SqlServerCe, Version=3.5.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91 |

2.2 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 <i>DataAdapter</i> . |
| 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*.

2.3 Interface portable

Les classes de chaque fournisseur varient, et donc par conséquent, 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 |
|-----------------------------|---|
| <code>IDataAdapter</code> | Permet de remplir et actualiser un objet <i>DataSet</i> et de mettre à jour une source de données. |
| <code>IDataReader</code> | Permet de lire un ou plusieurs flux de données en lecture seule à la suite de l'exécution d'une commande. |
| <code>IDataParameter</code> | Permet d'implémenter un paramètre pour une commande. |
| <code>IDbCommand</code> | Permet de donner une commande qui s'exécutera au moment de la connexion à une source de données. |
| <code>IDbConnection</code> | Représente une connexion unique avec une source de données. |
| <code>IDbDataAdapter</code> | Représente un jeu de méthodes qui permet d'exécuter des opérations sur des bases de données relationnelles (insertion, sélection, ...). |
| <code>IDbTransaction</code> | 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 :

```
'VB
Imports System.Data;

Imports System.Data.Common;

Imports System.Data.SqlClient;

//A rajouter

Sub Main()
    Dim requete, connexionString As String
    connexionString = "Data Source=.\SQLServeur;Initial
Catalog=DotNetFrance;Integrated Security=true;"
    requete = "SELECT * FROM Employee"

    Dim connexion As IDbConnection = New
SqlConnection(connexionString)
    Dim commande As IDbCommand = connexion.CreateCommand()

    commande.CommandText = requete
    commande.CommandType = CommandType.Text

    connexion.Open()
    Dim lire As IDataReader = 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"))
    End While
    connexion.Close()
    connexion.Dispose()

    Console.ReadLine()
End Sub
```

```
//c#

using System.Data;

using System.Data.Common;

using System.Data.SqlClient;

//A rajouter

static void Main(string[] args)
{
    string requete, connexionString;
    connexionString = @"Data Source=.\SQLServeur;Initial
Catalog=DotNetFrance;Integrated Security=true;";
    requete = "SELECT * FROM Employe";

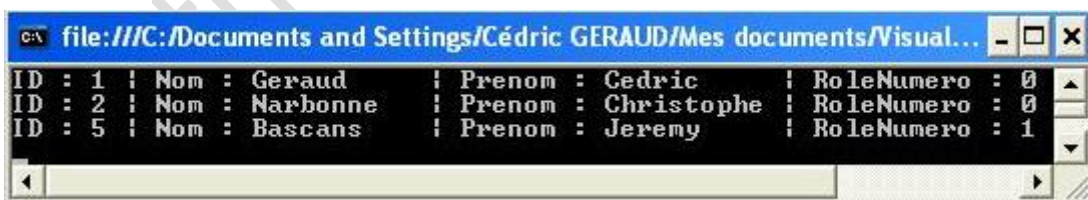
    IDbConnection connexion = new SqlConnection(connexionString);
    IDbCommand commande = connexion.CreateCommand();

    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();
}
```

Cela donnera quelque chose de semblable à :



```
file:///C:/Documents and Settings/Cédric GERAUD/Mes documents/Visual...
ID : 1 | Nom : Geraud | Prenom : Cedric | RoleNumero : 0
ID : 2 | Nom : Narbonne | Prenom : Christophe | RoleNumero : 0
ID : 5 | Nom : Bascans | Prenom : Jeremy | RoleNumero : 1
```

2.4 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 conséquent 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 entraînant 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 entraîner 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.

3 Etablir une connexion

3.1 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 » :

```
'VB  
  
Dim connexion As SqlConnection  
  
//C#  
  
SqlConnection connexion
```

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

```
'VB  
  
connexionString = "Data Source=.\SQLServeur;Initial  
Catalog=DotNetFrance;Integrated Security=true;"  
  
//C#  
  
connexionString = @"Data Source=.\SQLServeur;Initial  
Catalog=DotNetFrance;Integrated Security=true;"
```

Voici les différents paramètres disponibles dans une *ConnectionString* :

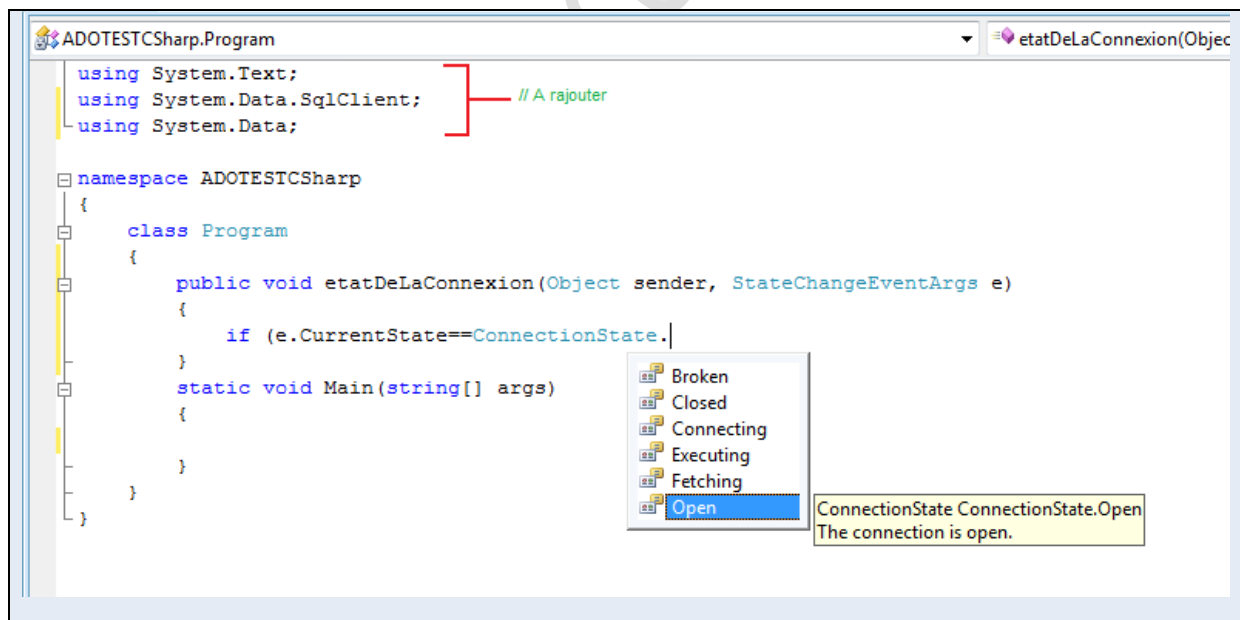
| Paramètre | Description |
|-----------|-------------|
|-----------|-------------|

| | |
|-----------------------|--|
| 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 est 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 peut 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. |

Voici, par exemple :



3.2 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 lorsque qu'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 <i>True</i> . |
| 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 <i>Max Pool Size</i> (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 (<i>True</i>) 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 <i>Connection Pool</i> . |
| ClearPool | SqlConnection et OracleConnection | Réinitialise une <i>Connection Pool</i> spécifique. |
| ReleaseObjectPool | OleDbConnection et Odbcconnection | Spécifie au pool de connexion qu'il peut être détruit lorsque la dernière connexion qu'il contenait a été détruite. |

3.3 Déconnexion

Pour couper la connexion entre l'application et la base de données, il suffit d'écrire :

```
'VB
nomConnexion.Close()

//C#
nomConnexion.Close();
```

4 Mode connecté

4.1 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 :

| <u>Propriétés</u> | |
|-------------------|---|
| Nom | Description |
| CommandText | Permet de définir l'instruction de requêtes SQL ou de procédures stockées à exécuter. Lié à la propriété <i>CommandType</i> . |
| 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 <i>CommandText</i> doit être exécutée. |
| Connection | Permet d'établir 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 <i>SqlTransaction</i> dans laquelle la <i>SqlCommand</i> s'exécute. |

Méthodes

| Nom | Description |
|------------------|---|
| 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. |

4.2 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 :

```
'VB
```

```
Dim commande As SqlCommand = New SqlCommand()  
commande.Connection = connexion  
commande.CommandText = "SELECT * FROM Employe"  
  
//C#  
  
SqlCommand commande= new SqlCommand();  
commande.Connection = connexion;  
commande.CommandText = "SELECT * FROM Employe";
```

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

```
'VB  
  
commande = New SqlCommand("SELECT * FROM Employe", connexion)  
  
//C#  
  
commande = new SqlCommand("SELECT * FROM Employe", connexion);
```

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

```
'VB  
  
Dim commande As SqlCommand = connexion.CreateCommand()  
commande.CommandText = "SELECT * FROM Employe"  
  
//C#  
  
SqlCommand commande = connexion.CreateCommand();  
commande.CommandText = "SELECT * FROM Employe";
```

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 pré-compilée dans le cache du serveur. Voici un exemple qui permet d'afficher toutes les informations d'un utilisateur de la table *Employe* :



```
'VB

Dim connexion As SqlConnection = New SqlConnection("Data
Source=.\SQLServeur;Initial Catalog=DotNetFrance;Integrated
Security=True")
Dim Command As SqlCommand = connexion.CreateCommand()
connexion.Open()

Dim id As String
Dim requete As String = "RecupInformation"

Command.CommandText = requete
Command.CommandType = CommandType.StoredProcedure

Console.WriteLine("Quel est l'id de la personne sur laquelle vous voulez
les informations ?")
id = Console.ReadLine()

Dim paramIdToKnow As SqlParameter = New SqlParameter("@IdToKnow", id)
Command.Parameters.Add(paramIdToKnow)

Dim lecture As IDataReader = Command.ExecuteReader()

While (lecture.Read())
    Console.WriteLine("Id : {0} Nom : {1} Prenom : {2} Role : {3}",
lecture("ID"), lecture.GetString(1), lecture.GetString(2),
lecture.GetInt32(3))
End While

connexion.Close()
connexion.Dispose()
```

```
//c#

SqlConnection connexion = new SqlConnection(@"Data
Source=.\SQLServeur;Initial Catalog=DotNetFrance;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 l'id de la personne 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["ID"], lecture.GetString(1), lecture.GetString(2),
lecture.GetInt32(3));
}

connexion.Close();
connexion.Dispose();
```

```
Quel est l'id de la personne sur laquelle vous voulez les informations ?
1
Id : 1 Nom : Geraud      Prenom : Cedric      Role : 0
```

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.

Voici, par exemple, un code permettant d'afficher la base de données Dot-Net France :

```
'VB
Dim connexion As SqlConnection = New SqlConnection("Data
Source=.\SQLServeur;Initial Catalog=DotNetFrance;Integrated
Security=True")
Dim Command As SqlCommand = connexion.CreateCommand()
Dim requete As String = "SELECT e.ID 'ID', e.Nom, e.Prenom, r.Nom FROM
Employe e, Role r WHERE(e.Role = r.ID) "
Command.CommandText = requete

connexion.Open()
Dim lire As SqlDataReader = Command.ExecuteReader()

Console.WriteLine("Lecture du DataReader" + vbNewLine + vbNewLine)
' Lit les informations de la base de données

While (lire.Read())
    Console.WriteLine("Id : {0} Nom : {1} Prenom : {2} Role : {3}",
lire("ID"), lire.GetString(1), lire.GetString(2), lire.GetString(3))
End While
' Permet d'afficher

connexion.Close()
connexion.Dispose()

//C#

SqlConnection connexion = new SqlConnection(@"Data
Source=.\SQLServeur;Initial Catalog=DotNetFrance;Integrated
Security=True");
SqlCommand command = connexion.CreateCommand();
string requete = "SELECT e.ID 'ID', e.Nom, e.Prenom, r.Nom FROM Employe
e, Role r WHERE(e.Role = r.ID) ";
command.CommandText = requete;

connexion.Open();
SqlDataReader lire = command.ExecuteReader();
// Lit les informations de la base de données

Console.WriteLine("Lecture du DataReader \n\n");

while (lire.Read())
{
    Console.WriteLine("Id : {0} Nom : {1} Prenom : {2} Role : {3}",
lire["ID"], lire.GetString(1), lire.GetString(2), lire.GetString(3));
}
// Permet d'afficher

connexion.Close();
connexion.Dispose();
```

Vous devriez obtenir un affichage semblable à :

```
Lecture du DataReader
Id : 1 Nom : Geraud      Prenom : Cedric      Role : Stagiaire
Id : 2 Nom : Narbonne   Prenom : Christophe  Role : Stagiaire
Id : 5 Nom : Bascans    Prenom : Jeremy      Role : Manager
```

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

4.3 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.

4.4 Les types de paramètres

La modification des informations contenues 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.

4.5 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.

Voici un exemple permettant d'ajouter un utilisateur dans la base de données DotNetFrance :

```
'VB
Dim connexion As SqlConnection = New SqlConnection("Data
Source=.\SQLServeur;Initial Catalog=DotNetFrance;Integrated
Security=True")
Dim Command As SqlCommand = connexion.CreateCommand()
Dim nom, prenom, requete, role As String

Console.WriteLine("Entrez les données suivantes :" + vbNewLine)
' Ecrit directement dans la console
Console.WriteLine("Nom : ")
nom = Console.ReadLine()
' Permet de lire la valeur donnée
Console.WriteLine(vbNewLine + "Prenom : ")
prenom = Console.ReadLine()
Console.WriteLine(vbNewLine + "Role (0 Stagiaire | 1 Manager) : ")
role = Console.ReadLine()
requete = "INSERT INTO Employe VALUES(@nom, '" + prenom + "'," + role +
")"
Dim param As SqlParameter = New SqlParameter("@nom", nom)
' Permet de paramétrer "nom"
Command.Parameters.Add(param)
' Ajoute le paramètre param à la collection Parameters
Command.CommandText = requete

connexion.Open()
Dim nbrEnregistrementAffecte As String =
Command.ExecuteNonQuery().ToString()
Console.WriteLine(vbNewLine + nom + " " + prenom + " " + role + " a ete
ajoute.")
Console.WriteLine("Il y a " + nbrEnregistrementAffecte + "
enregistrement(s) affecte par la requete")
```

```
//c#

SqlConnection connexion = new SqlConnection(@"Data
Source=.\SQLServeur;Initial Catalog=DotNetFrance;Integrated
Security=True");
SqlCommand command = connexion.CreateCommand();
string nom, prenom, requete;
int role;

Console.WriteLine("Entrez les données suivantes :\n");
// Ecrit directement dans la console
Console.WriteLine("Nom : ");
nom = Console.ReadLine();
// Permet de lire la valeur donnée
Console.WriteLine("\nPrenom : ");
prenom = Console.ReadLine();
Console.WriteLine("\nRole (0 Stagiaire | 1 Manager) : ");
role = int.Parse(Console.ReadLine());
requete = "INSERT INTO Employe VALUES(@nom, '" + prenom + "'," + role +
")";
SqlParameter param = new SqlParameter("@nom", nom);
// Permet de paramétrer "nom"
command.Parameters.Add(param);
// Ajoute le paramètre param à la collection Parameters
command.CommandText = requete;

connexion.Open();
int nbrEnregistrementAffecte = command.ExecuteNonQuery();
Console.WriteLine("\n" + nom + " " + prenom + " " + role + " a ete
ajoute.");
Console.WriteLine("Il y a " + nbrEnregistrementAffecte + "
enregistrement(s) affecte par la requete");
```

```
Entrez les données suivantes :
Nom :
Test
Prenom :
Test
Role <0 Stagiaire | 1 Manager> :
0
Test Test 0 a ete ajoute.
Il y a 1 enregistrement(s) affecte par la requete
```

Remarque : Vous pouvez aussi ajouter des paramètres à vos Command Object en saisissant la méthode `GetCostCommand`.

4.6 Les BLOBs

Les *BLOBs* dans une base de données ne sont pas de simples données de types chaînes 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 conséquent leur utilisation est plus complexe.

La taille d'un *BLOB* peut dépasser plusieurs Go et par conséquent 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*.

4.7 Le *DataReader*

Le *DataReader* permet un accès en lecture seule à des enregistrements, c'est-à-dire qu'il est impossible de revenir en arrière sur les enregistrements lus. Il n'a pas été créé que pour la lecture pure et simple de données. Le *DataReader* doit toujours être associé à une connexion active, c'est-à-dire qu'il ne peut pas se déconnecter, effectuer quelque chose puis se reconnecter pour une mise à jour.

Il existe plusieurs *DataReader* suivant le fournisseur utilisé, par exemple nous avons *SqlDataReader* ou encore *OracleDataReader*.

Le *DataReader* comprend plusieurs méthodes : *GetBytes*, *GetChars* ou *GetString*. *GetString* servent pour les données caractères ; *GetBytes* et *GetChars* permettent de retourner une valeur de type *long*, qui représente le nombre de caractères ou d'octets retournés.

4.8 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.

4.9 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 :

```
'VB
'Création d'une transaction
Dim transaction As New SqlTransaction

'Définit la transaction à votre connexion
Transaction = VotreConnexion.BeginTransaction()

//C#
'Création d'une transaction
SqlTransaction transaction

'Définit la transaction à votre connexion
transaction = VotreConnexion.BeginTransaction() ;
```

De plus les transactions reposent sur le principe de quatre caractéristiques appelé ACID qui apporte plus de clarté sur la définition d'une transaction :

- **Atomicité**, qui signifie que la mise à jour de la base de données doit être totale ou nulle, c'est le principe du "tout ou rien".
- **Cohérence**, qui indique que les modifications apportées doivent être valides.
- **Isolation**, qui définit que les transactions lancées au même moment ne doivent pas s'interférer entre elles.
- **Durabilité**, qui assure que toutes les transactions sont lancées de manière définitive.

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*.

4.9.1 Niveau d'isolations des transactions

Les niveaux d'isolation vous permettent de gérer les problèmes d'intégrité des données ainsi que des accès simultanés à celles-ci par le biais des transactions. Vous trouverez ci-dessous la liste des propriétés *IsolationLevel* associés à l'objet *Transaction* :

| Nom | Description |
|-----------------|---|
| Chaos | Les modifications en attente de transactions très isolés ne peuvent être écrasées. |
| ReadCommitted | Les verrouillages partagés sont maintenus pendant que les données sont en lecture pour éviter tout défaut, mais les données sont modifiables durant la transaction, entraînant une lecture non répétable ou des données "fantôme". |
| ReadUncommitted | Il n'y a pas de verrouillage entraînant la possibilité de tout défaut de lecture. |
| RepeatableRead | Toutes les données utilisées dans une requête sont verrouillées. Par conséquent les lectures ne sont pas répétables mais des données "fantôme" peuvent exister et d'autres utilisateurs ne peuvent mettre à jour les données. |
| Serializable | Les verrouillages sont placés sur toutes les données qui sont utilisées dans une requête, ce qui empêche d'autres utilisateurs de mettre à jour les données. |
| Snapshot | Réduit le blocage par le stockage de données qu'une seule application peut lire pendant qu'une autre est train de les modifier. Cela indique que d'une seule transaction vous ne pouvez voir les modifications faites dans d'autres transactions. |
| Unspecified | Aucun niveau ne peut être déterminé. Ainsi si le niveau d'isolation est défini sur celui-ci alors la transaction exécute selon le niveau d'isolation par défaut du sous-jacent du type de base de données. |

4.9.2 Les transactions Distribuées

En plus de créer de nouvelles transactions et de définir des niveaux d'isolation de celles-ci, vous pouvez configurer votre connexion pour l'engager dans une transaction distribuée. Une transaction distribuée est une transaction qui s'étend sur de nombreuses ressources, telles que de multiples bases de données SQL.

Exemple de mise en place d'une transaction distribuée :

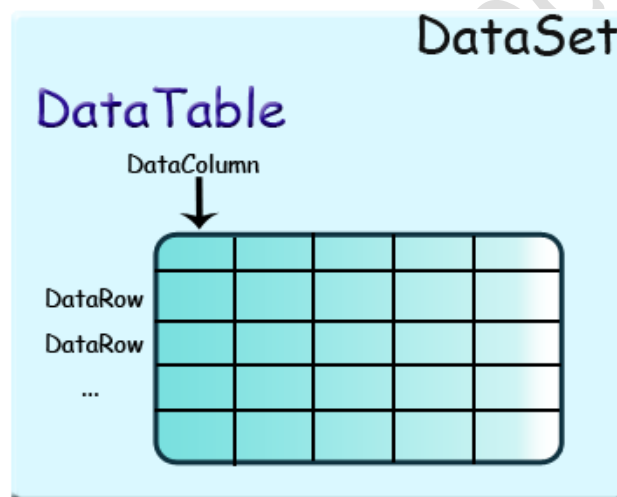
```
'VB
VotreConnexion.EnlistTransaction(activeTransaction)

//C#
VotreConnexion.EnlistTransaction(activeTransaction) ;
```

5 Mode déconnecté

5.1 Le DataSet

Le *DataSet* est stocké dans l'espace de nom *System.Data*. C'est un cache de données en mémoire, c'est-à-dire qu'il permet de stocker temporairement des données utilisées dans votre application. Le *DataSet* contient la collection d'objets *DataTable* qui peuvent être liés avec les objets *DataRelation*. Dans le cas du mode déconnecté, cet objet va nous permettre d'importer la partie désirée de la base de données (fonction de la requête de sélection) en local. Ainsi grâce à des objets nécessaires à la connexion classique (*commande select*, *connections string...*) et un *DataAdapter*, nous pourrions relier ("Binder") un *DataSet* sur une base de donnée (en lecture et en écriture grâce à une méthode de mise à jour de la base de donnée).



Grâce au mot clé *foreach*, on peut parcourir des *DataSet* ou les requêter en *Linq*.

La création d'un *DataSet* peut se faire via plusieurs méthodes :

- On peut le créer par programmation en créant des objets *DataTable*, *DataRelation* (facultatif). Il suffit ensuite de remplir là où les base(s) de données.
- Vous pouvez aussi utiliser le *Designer DataSet* et l'assistant de configuration de *Data Source* afin de créer un objet *DataSet*. Pour cela vous devez mettre en place le processus de connexion à des données en sélectionnant une base de données disponibles, qui permettra de mettre en place un *DataSet* typé. Ainsi cela génère le code nécessaire pour cette création.
- Vous pouvez aussi glisser directement un *DataSet* depuis la *Toolbox* sur votre *Form* (cas pour le WinForm)

Il existe deux types de *DataSet* :

- *Typed DataSet* : Cela permet de créer une instance d'une classe *DataSet* déjà typée dans votre projet.
- *Untyped DataSet* : Cela permet de créer une instance d'une classe *DataSet* non-typé de la classe *System.Data.DataSet*.

Voici les éléments d'un *DataSet* :

| Objet | Description |
|-------------------|---|
| <i>DataTable</i> | Correspond à une table. Contient une collection de <i>DataColumn</i> et de <i>DataRow</i> . |
| <i>DataColumn</i> | Représente une colonne de la table. |
| <i>DataRow</i> | Correspond à un enregistrement de la table. |

5.2 Configurer un *DataSet*

5.2.1 Créer un *DataSet*

Pour créer un *DataSet*, il faut en premier lieu les instancier (vous pouvez aussi définir leur nom) :

```
'VB
Dim DotNetFranceDataSet As New DataSet()

//C#
DataSet DotNetFranceDataSet = new DataSet();
```

5.2.2 Ajouter des *DataTable*

Après avoir créé votre *DataSet*, vous pouvez lui ajouter des *DataTable* comme ceci :

```
'VB
Dim Employe As New DataTable
Dim Role As New DataTable

DotNetFranceDataSet.Tables.Add(Employe)
DotNetFranceDataSet.Tables.Add(Role)
```

```
//c#

DataTable Employe = new DataTable();
DataTable Role = new DataTable();

DotNetFranceDataSet.Tables.Add(Employe);
DotNetFranceDataSet.Tables.Add(Role);
```

5.2.3 Le *DataRelation*

L'objet *DataRelation* représente le lien entre plusieurs tables. Dans un *DataSet*, il permet la relation entre les colonnes de plusieurs *DataTable*. Par exemple, nous pouvons lier les colonnes ID et Role de nos tables *Employe* et *Role* :

```
'VB

Dim Employe As New DataRelation ("EmployeRole",
Employe.ParentColumns("Role"), Role.Columns("ID"))
DotNetFranceDataSet.Relations.Add(Employe)

//c#

DataRelation Employe = new DataRelation("EmployeRole",
Employe.ParentColumns["Role"], Role.Columns["ID"]);
DotNetFranceDataSet.Relations.Add(Employe);
```

Ensuite, grâce à cette relation, vous pouvez naviguer entre les colonnes de vos tables. Le *DataRow* propose deux méthodes : le *GetParentRow* et le *GetChildRows* :

| Méthode | Description |
|----------------------|---|
| <i>GetParentRows</i> | Permet d'obtenir les lignes parentes d'un <i>DataRow</i> attaché à un <i>DataRelation</i> . |
| <i>GetChildRows</i> | Permet d'obtenir les lignes enfants d'un <i>DataRow</i> attaché à un <i>DataRelation</i> . |

Par exemple, nous pouvons afficher les ID de notre précédente relation « *EmployeRole* » :

```
'VB

Dim IdCorrespondant As DataRow = ID.GetParentRow("FK_Employe_Role")

//c#

DataRow IdCorrespondant = ID.GetParentRow("FK_Employe_Role");
```

5.2.4 Fusionner deux *DataSet*

La méthode *Merge* permet de fusionner deux *DataSet* entre eux. On prend un *DataSet* source qu'on fusionne avec un *DataSet* ciblé. Pour des tables identiques, ce sont les données du *DataSet* source qui remplaceront les données du *DataSet* ciblé. Lorsqu'il s'agit de deux *DataSet* différents, vous pouvez paramétrer la méthode *Merge* avec la propriété *MissingSchemaAction*. Voici les différentes valeurs de ce paramètre :

| Valeur | Description |
|------------|--|
| Add | Permet d'ajouter les objets contenus dans le <i>DataSet</i> source dans le <i>DataSet</i> ciblé. |
| AddWithKey | Permet d'ajouter les objets contenus dans le <i>DataSet</i> source dans le <i>DataSet</i> ciblé avec les clés primaires. |
| Error | Une exception sera levée si le mappage d'une colonne dans les <i>DataSet</i> source et ciblé est manquante. |
| Ignore | Toutes les colonnes du <i>DataSet</i> source supplémentaire sont ignorées. |

5.2.5 Copier un *DataSet*

Vous pouvez par exemple vouloir modifier pour une série de test un *DataSet*. Pour cela, il est très simple de copier un *DataSet*. Il suffit d'utiliser la méthode *Copy* :

```
'VB
Dim CopieDuDataSet As New DataSet
CopieDuDataSet = DataSetDorigine.Copy

//C#
DataSet CopieDuDataSet = new DataSet();
CopieDuDataSet = DataSetDorigine.Copy();
```

5.3 Configurer une *DataTable*

5.3.1 Créer des colonnes contenant des expressions

Dans une base de données, vous pouvez avoir besoin de créer des colonnes contenant des expressions. Ceci fonctionne un peu comme avec un tableau Microsoft Excel, vous pouvez créer des expressions dans une colonne qui permet de calculer une somme, une multiplication ou autre plus complexe. Il existe pour cela l'objet *Expression* :

```
'VB
Dim SommeColonne As New DataColumn("Somme", GetType(Decimal))
SommeColonne.Expression = ("Colonne1 + Colonne2")
DotNetFranceDataSet.MaTable.Columns.Add(SommeColonne)

//C#
DataColumn SommeColonne = new DataColumn("Somme",
Type.GetType("decimal"));
SommeColonne.Expression = ("Colonne1 + Colonne2");
DotNetFranceDataSet.MaTable.Columns.Add(SommeColonne);
```

Remarque : Si vous gérez des nombres à virgules, changez le paramètre *Decimal* en *Double*.

5.3.2 L'incrémentation dans une *DataTable*

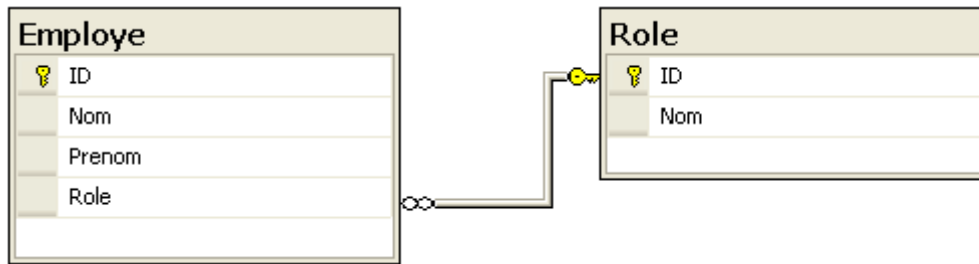
L'auto-incrémentation est le fait d'incrémenter (augmenter) un nombre. On peut l'incrémenter avec un nombre positif ou un nombre négatif. L'incrémentation se fait sur une colonne. Pour incrémenter, il faut mettre à *True* la propriété *AutoIncrement*. La propriété *AutoIncrementSeed* définit le nombre de départ et *AutoIncrementStep* définit le pas de l'incrémentation (c'est celui qui peut être positif ou négatif).

5.3.3 Les contraintes dans une *DataTable*

Une *DataTable*, à l'image des contraintes dans les bases de données relationnelles, peut également avoir des contraintes appliquées à certaines de ses colonnes.

Rappel : En SQL, une clé primaire permet d'identifier de manière unique chaque ligne d'une table. Les champs associés à une clé primaire ont une valeur unique dans une table. Une clé étrangère permet de faire une référence vers une autre colonne de la table ou vers une

autre table, possédant obligatoirement une clé primaire. Par exemple cette base de données :



Il existe les mêmes deux types de clés : les clés étrangères et les clés primaires.

- Les clés primaires : Pour installer une clé primaire à une valeur unique, il suffit de créer l'instance de la classe *UniqueConstraint* et d'assigner la colonne avec la valeur unique au constructeur.
- Les clés étrangères : Afin d'ajouter une clé à une ou plusieurs colonnes voir plusieurs tables parents aux colonnes enfants il faut faire l'instance de la classe *ForeignKeyConstraint*. Cette classe possède les propriétés *DeleteRule* et *UpdateRule* qui permet de définir le comportement. Il en existe quatre valeurs :

| Valeur | Description |
|------------|---|
| Cascade | Les modifications se font sur toutes les lignes liées par la clé. |
| None | Les modifications ne se font pas sur les lignes liées par la clé. |
| SetDefault | La valeur par défaut est donnée dans les lignes liées. |
| SetNull | La valeur est modifiée par DBNull. |

5.4 Le *DataAdapter*

L'objet *DataAdapter* permet de relier un *DataSet* à une base de données. En revanche, le *DataAdapter* change suivant le fournisseur, c'est-à-dire, par exemple, pour une base de données SQL, ce sera *SqlDataAdapter*. C'est grâce à cela que votre application pourra communiquer avec la base de données et par exemple mettre à jour celle-ci.

5.4.1 Créer un *DataAdapter*

Pour créer un *DataAdapter*, il faut tout d'abord instancier celui correspondant à votre fournisseur. Il a besoin d'une commande pour fonctionner. Par exemple :

```
'VB
Dim DotNetFranceAdapter As New SqlDataAdapter("SELECT * FROM Employe",
connexion)

//C#
SqlDataAdapter DotNetFranceAdapter = new SqlDataAdapter("SELECT * FROM
Employe", connexion);
```

5.4.2 Créer des commandes pour un *DataAdapter*

Il est possible d'ajouter plusieurs commandes à un *DataAdapter*. Pour cela, il suffit de créer votre commande puis de l'ajouter à votre *DataAdapter*. Par exemple, voici une commande associée à un *DataAdapter* permettant de modifier le prénom de la table Employe :

```
'VB
Dim ModifierNom As New SqlCommand("UPDATE Employe SET Prenom = 'Jeremie'
WHERE Prenom='Jeremy'")
DotNetFranceAdapter.ModifierNom = ModifierNom

//C#
SqlCommand ModifierNom = new SqlCommand("UPDATE Employe SET Prenom =
'Jeremie' WHERE Prenom='Jeremy'");
DotNetFranceAdapter.ModifierNom = ModifierNom;
```

5.4.3 Le *MissingMappingAction*

Cette propriété permet de régler certains conflits entre un *DataSet* et une base de données. En effet des conflits peuvent arriver si on tente par exemple de remplir un *DataSet* (avec une base de données) dans lequel il manque soit des colonnes ou des tables.

L'énumération *MissingMappingAction* possède ces valeurs possibles :

| Valeur | Description |
|-------------|--|
| Error | Permet de lever l'exception <i>InvalidOperationException</i> si une colonne est manquante. |
| Ignore | Permet de retourner une valeur NULL si une table ou une colonne est manquante. |
| Passthrough | Permet de créer la colonne et/ou la table manquante. |

5.4.4 Le *MissingSchemaAction*

Cette propriété permet aussi de régler les conflits entre un *DataSet* et une base de données. Il permet de spécifier où et comment ajouter des données dans des *DataTable* et *DataColumn* manquante. Voici les valeurs de l'énumération *MissingMappingAction* :

| Valeur | Description |
|------------|--|
| Add | Permet de créer la colonne et/ou la table manquante. |
| AddWithKey | Permet d'ajouter les clés nécessaires aux colonnes. |
| Error | Permet de lever l'exception <i>InvalidOperationException</i> si une colonne est manquante. |
| Ignore | Permet d'ignorer les colonnes en trop. |

5.5 Gérer une *DataTable*

5.5.1 Ajouter des données

Afin d'ajouter des données dans une *DataTable*, il faut tout d'abord créer un nouveau *DataRow*. Après avoir créé votre ligne, vous pouvez y ajouter vos valeurs. Par exemple, nous pouvons ajouter un nouvel utilisateur à notre base de données avec un ID, Nom, Prenom et le rôle :

```
'VB
DotNetFranceDataSet.Tables("Employe").Rows.Add(5, "Bascans", "Jérémy", 1)

//C#
DotNetFranceDataSet.Tables["Employe"].Rows.Add(5, "Bascans", "Jérémy", 1);
```

5.5.2 Gérer des données

Vous pouvez aussi éditer ou supprimer des données dans votre *DataTable* comme ceci :

| Langage | Modifier | Supprimer |
|---------|----------------------------|--------------------|
| C# | MaLigne["ID"] = "Numero" ; | MaLigne.Delete() ; |
| VB | MaLigne("ID") = "Numero" | MaLigne.Delete |



Le *RowState* vous permet de gérer le statut d'une ligne. Par défaut, il se trouve sur *Unchanged*. Voici les valeurs possibles de cette énumération :

| Valeur | Description |
|-----------|--|
| Added | Permet à la ligne de s'ajouter dans la collection <i>DataRowCollection</i> . |
| Deleted | Permet de supprimer la ligne en utilisant la méthode <i>Delete</i> de <i>DataRow</i> . |
| Detached | Permet de créer la ligne sans qu'elle n'appartienne à la collection <i>DataRowCollection</i> . |
| Modified | Permet de modifier la ligne. |
| Unchanged | Permet de ne pas modifier la ligne. |

5.5.3 Les évènements d'une *DataTable*

La *DataTable* possède ses propres évènements :

| Evènement | Description |
|----------------|--|
| ColumnChanged | Se déclenche lorsqu'une valeur a été changée dans un <i>DataRow</i> d'une <i> DataColumn</i> . |
| ColumnChanging | Se déclenche lorsqu'une valeur est en train d'être changée dans un <i>DataRow</i> d'une <i> DataColumn</i> . |
| Disposed | Est appelée lorsque la méthode <i>Dispose</i> est appelée. |
| Initialized | Se déclenche après que la <i>DataTable</i> soit initialisée. |
| RowChanged | Se déclenche lorsqu'un <i>DataRow</i> a bien été modifié. |
| RowChanging | Se déclenche lorsqu'un <i>DataRow</i> est en modification. |
| RowDeleted | Se déclenche lorsque qu'une ligne a été supprimée. |
| RowDeleting | Se déclenche lorsque qu'une ligne est en train d'être supprimée. |
| TableCleared | Se déclenche quand une <i>DataTable</i> a été vidée. |
| TableClearing | Se déclenche quand une <i>DataTable</i> se vide. |
| TableNewRow | Se déclenche lorsqu'un <i>DataRow</i> est ajouté. |

5.6 Le *DataView*

Un *DataView* permet d'avoir une « vue » d'une *DataTable*. Cela permet aussi de pouvoir la trier, la modifier. La fonction principale de cet objet est de relier une base de

données aux applications WinForm et WebForm.

5.6.1 Créer un *DataGridView*

Afin de créer un *DataGridView*, il suffit de créer l'instance de celle-ci et d'y placer le nom de la *DataTable* souhaitée. Par exemple :

```
'VB
Dim MonDataGridView As New DataGridView(DotNetFranceDataSet.Tables("Employe"))
Dim MonDataGridView As DataGridView =
DotNetFranceDataSet.Tables["Employe"].DefaultView

//C#
DataGridView MonDataGridView = new
DataGridView(DotNetFranceDataSet.Tables["Employe"]);
DataGridView MonDataGridView = DotNetFranceDataSet.Tables["Employe"].DefaultView;
```

5.6.2 Gérer un *DataGridView*

La propriété *Sort* du *DataGridView* permet de le trier. On peut le trier de deux façons : ASC (croissant) et DESC (décroissant). La façon croissante est par défaut utilisée par le *DataGridView*.

```
'VB
MonDataGridView.Sort = "Nom ASC"

//C#
MonDataGridView.Sort = "Nom ASC";
```

Le *DataGridViewRow* permet de modifier les lignes d'une table. Par exemple :

```
'VB
DataGridViewRow("Nom") = "Jeremie"

//C#
DataGridViewRow("Nom") = "Jeremie";
```

L'évènement principal d'un *DataView* est le *ListChanged*. Cet évènement se déclenche lorsque qu'il y a un changement dans une *DataTable* ou *DataRelation* relié à ce *DataView*.

5.7 Exemples

5.7.1 Remplissage automatique / Lecture de données

```
'VB
Dim requete, connexionString As String
connexionString = "Data Source=.\SQLServeur;Initial Catalog=DotNetFrance;Integrated
Security=true;"
requete = "SELECT * FROM Employe"

Dim ds As DataSet = New DataSet()
Dim da As IDbDataAdapter = New SqlDataAdapter()

Dim connexion As IDbConnection = New SqlConnection(connexionString)
Dim commande As IDbCommand = connexion.CreateCommand()
commande.CommandText = requete
' On choisit la commande qui sera exécuté par le DataAdapter et on remplit le
DataSet
da.SelectCommand = commande
da.Fill(ds)

'Affichage du DataSet
Console.WriteLine("Nom de la table : {0} | Il possède {1}
enregistrement(s)" + vbNewLine + vbNewLine + vbNewLine, ds.Tables(0),
ds.Tables("Table").Rows.Count) 'On affiche les informations que l'on tire
du DataSet

For Each colonne As DataColumn In ds.Tables(0).Columns 'Affichage des noms des
colonnes
    Console.WriteLine(colonne.ColumnName + vbTab)
Next
Console.WriteLine(vbNewLine + vbNewLine)

For Each ligne As DataRow In ds.Tables(0).Rows 'Affichage des enregistrements
```

```
//c#

string requete, connexionString;
connexionString = @"Data Source=.\SQLServeur;Initial
Catalog=DotNetFrance;Integrated Security=true;";
requete = "SELECT * FROM Employe";

DataSet ds = new DataSet();
IDataAdapter da = new SqlDataAdapter();

IbConnection connexion = new SqlConnection(connexionString);
IbCommand commande = connexion.CreateCommand();
commande.CommandText = requete;
commande.CommandType = CommandType.Text;

// On choisit la commande qui sera executé par le DataAdapter et on remplit le
DataSet
da.SelectCommand = commande;
da.Fill(ds);

//Affichage du DataSet
Console.WriteLine("Nom de la table : {0} | Il possède {1} enregistrement(s)
\n\n\n", ds.Tables[0],ds.Tables["Table"].Rows.Count); //On affiche les informations
que l'on tire du DataSet

foreach (DataColumn colonne in ds.Tables[0].Columns) //Affichage des noms des
colonnes
{
    Console.Write(colonne.ColumnName + "\t");
}
Console.WriteLine("\n\n");

foreach (DataRow ligne in ds.Tables[0].Rows) //Affichage des enregistrements
{
    foreach (DataColumn colonne in ds.Tables[0].Columns)
    {
        Console.Write(ligne[colonne.ColumnName] + "\t");
    }
    Console.WriteLine("\n");
}
Console.ReadLine();
```

Nom de la table : Table ! Il possède 3 enregistrement(s)

| ID | Nom | Prenom | Role |
|----|----------|------------|------|
| 1 | Geraud | Cedric | 0 |
| 2 | Narbonne | Christophe | 0 |
| 5 | Bascans | Jeremy | 1 |

5.7.2 Remplissage manuel du DataSet

```
'VB
Module Module1
    Sub Main()
        Dim ds As DataSet = New DataSet("Dotnet-France")
        Dim stagiaires As DataTable = New DataTable("Stagiaires")
        ' cette table contient trois champs à renseigner pour chaque enregistrement
        stagiaires.Columns.Add("ID", GetType(Integer)) ' l'id
        stagiaires.Columns.Add("Nom", GetType(String)) ' le nom
        stagiaires.Columns.Add("Description", GetType(String)) ' description
        ' méthode du dictionnaire
        Dim dnom As Dictionary(Of String, String) = New Dictionary(Of String,
String) ()
        dnom.Add("Cédric", "Stagiaire")
        dnom.Add("Christophe", " Stagiaire ") ' pour chaque paire
d'éléments nous allons ajouter des enregistrements
        For Each elem As KeyValuePair(Of String, String) In dnom
            stagiaires.Rows.Add(Class1.StagiaireID, elem.Key, elem.Value)
        Next
        ' des listes
        Dim nom() As String = {"Jérémy", "Cédric"}
        Dim desc As String() = {"Manager ", "Manager"} ' s'il y a beaucoup
de champs pour des raisons de lisibilités il peut etre préférable
        ' de ranger chaque liste dans un Dictionary<string,List<string>>
        For i = 0 To nom.Length - 1 Step 1
            stagiaires.Rows.Add(Class1.StagiaireID, nom(i), desc(i))
        Next
        ' et comme vous l'aurez compris :remplissage brut
        ' (s'il n'y a que très peu d'entrée peut etre une bonne solution)
        stagiaires.Rows.Add(Class1.StagiaireID, "Paul", "Stagiaire")
        stagiaires.Rows.Add(Class1.StagiaireID, "Quentin", "Stagiaire")
        ' on peut ranger notre table dans notre DataSet
        ds.Tables.Add(stagiaires)
        ' donc nous allons afficher pour prouver le fonctionnement
        ' les données dans un terminal
        For Each row As DataRow In ds.Tables("stagiaires").Rows
            For Each col As DataColumn In stagiaires.Columns
                Console.Write(row.Item(col.ColumnName).ToString() + vbTab)
            Next
            ' retour à la ligne entre chaque enregistrement.
            Console.WriteLine()
        Next
        Console.ReadLine()
    End Sub

    Public Class Class1
        ' on crée une valeur id avec accesseur pour l'auto incrementation
        Private Shared _stagiaireid As Int32 = 0
        Public Shared ReadOnly Property StagiaireID() As Int32
            Get
                _stagiaireid += 1
                Return _stagiaireid
            End Get
        End Property
    End Class
End Module
```




```
//c#

class MainClass
{
    public static void afficheTable(DataSet ds, string tableName)
    {
        foreach (DataColumn col in ds.Tables[tableName].Columns)
        {
            Console.Write(col.ColumnName + "\t");
        }
        Console.WriteLine();
        foreach (DataRow row in ds.Tables[tableName].Rows)
        {
            foreach (DataColumn col in ds.Tables[tableName].Columns)
            {
                Console.Write(row[col.ColumnName].ToString() + "\t");
            }
            Console.WriteLine();
        }
        Console.WriteLine("\n");
    }
    private static int _stagiaireid = 0;
    public static int StagiaireID { get { return _stagiaireid++; } }

    public static void Main(string[] args)
    {
        DataSet ds = new DataSet("dotnet-france");
        DataTable stagiaires = new DataTable("stagiaires");
        // cette table contient trois champs à renseigner pour
        // chaque enregistrement l'id, le nom et la description
        stagiaires.Columns.Add("id", typeof(int));
        stagiaires.Columns.Add("nom ", "text".GetType());
        stagiaires.Columns.Add("description", typeof(string));
        // Rangement dans une collection générique
        Dictionary<string, string> dnom = new Dictionary<string, string>();
        dnom.Add("Cédric", "Stagiaire");
        dnom.Add("Christophe", "Stagiaire");

        foreach (KeyValuePair<string, string> elem in dnom)
        {
            stagiaires.Rows.Add(StagiaireID, elem.Key, elem.Value);
        }

        // dans des listes
        string[] nom = { "Jérémy", "Cédric" };
        string[] desc = { "Manager", "Manager" };

        for (int i = 0; i < nom.Length; i++)
        {
            stagiaires.Rows.Add(StagiaireID, nom[i], desc[i]);
        }

        // remplissage brut
        stagiaires.Rows.Add(StagiaireID, "Paul", "Stagiaire");
        stagiaires.Rows.Add(StagiaireID, "Quentin", "Stagiaire");

        ds.Tables.Add(stagiaires);

        afficheTable(ds, "stagiaires");
    }
}
```

```
id      nom      description
0       Cédric   Stagiaire
1       Christophe Stagiaire
2       Jérémy   Manager
3       Cédric   Manager
4       Paul     Stagiaire
5       Quentin  Stagiaire
```

6 Conclusion

Vous avez terminé ce chapitre sur la gestion de bases de données avec ADO.NET. Comme vous avez pu le constater, le mode connecté et déconnecté ont chacun des utilités ainsi qu'une utilisation différente. Pour de nouveaux exemples et pour compléter votre formation, veuillez regarder le webcast associé à ce chapitre.

L'équipe ADO.NET.