afpa

gagnez en compétences

Secteur Tertiaire Informatique
Filière étude - développement

Développer des composants d'interface

Accès aux données :

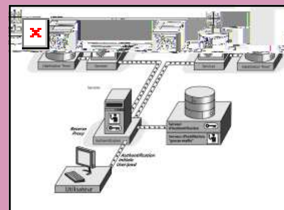
Le mode connecté

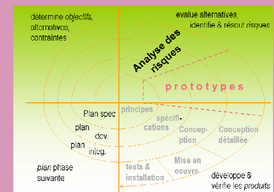
Accueil

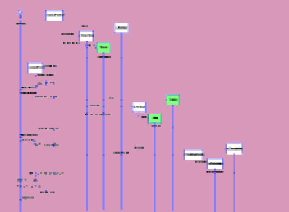
Apprentissage

Période en entreprise

Evaluation







Code barre

SOMMAIRE

I INTRODUCTION 4
I.1 Mode connecté et mode déconnecté..4
I.2 Les fournisseurs de données (providers)..5
I.3 Schéma général d'ADO.Net 6
II LE MODELE DE PROGRAMMATION ADO.NET.................................8
II.1 La classe CONNEXION...8
II.1.1 Stocker les chaînes de connexion 14
II.1.2 Extraction des chaînes de connexions 15
II.1.3 Pool de connexions 16
II.1.4 Obtenir des informations de schéma relationnel d'une base de données..16
II.1.5 Intercepter des évènements issus d'une connexion 16
II.2 La classe COMMANDE 17
II.2.1 Obtenir des données avec l'objet DataReader 19
II.2.2 Obtenir une valeur de la base de données 23
II.2.3 Mettre à jour les données avec des requêtes SQL.........................24
II.2.4 Utiliser des procédures stockées...25
II.2.5 Paramétrer requêtes et procédures stockées.................................25
II.3 Travailler avec des transactions locales 28
II.3.1 Gérer les transactions locales dans ADO.Net 1.X et ADO 2.0 28
II.3.2 Gérer les transactions locales dans ADO 2.0 29
II.3.3 La gestion des erreurs 31
III ANNEXES 33

I INTRODUCTION

L'appellation ADO.NET englobe à la fois l'ensemble des classes du *framework .NET* utilisées pour manipuler les données contenues dans des SGBD relationnels, et la philosophie d'utilisation de ces classes.

Avant ADO.NET, la technologie *ADO (ActiveX Data Object)* constituait l'ensemble des classes qu'il fallait utiliser pour accéder aux bases de données dans les environnements Microsoft. Malgré son nom, ADO.NET est beaucoup plus qu'un successeur de ADO. Bien que ces deux technologies aient une finalité commune, de profondes différences les séparent, notamment parce qu'ADO.NET est beaucoup plus complet.

I.1 MODE CONNECTE ET MODE DECONNECTE

Les notions de *mode déconnecté* et de *mode connecté* décrivent la façon dont une application travaille avec une base de données. Il faut d'abord introduire la notion de *connexion* avec une base de données. Une connexion est une ressource qui est initialisée et utilisée par une application pour travailler avec une base de données. Une connexion avec une base de données peut prendre les états ouvert et fermé. Si une application détient une connexion ouverte avec une base de données, on dit qu'elle est connectée à la base. Une connexion est en général initialisée à partir d'une chaîne de caractères qui contient des informations sur le type de SGBD supportant la base de données et/ou sur la localisation physique de la base de données.

Lorsqu'une application travaille avec ADO .NET, de nombreux cas de figures sont possibles :

- Une application travaille en mode connecté si elle charge des données de la base au fur et à mesure de ses besoins. L'application reste alors connectée à la base. L'application envoie ses modifications sur les données à la base au fur et à mesure qu'elles surviennent.
- Une application travaille en mode déconnecté dans les cas suivants :
  - ⇒ Si l'application charge des données de la base, qu'elle se déconnecte de la base, qu'elle exploite et modifie les données, qu'elle se reconnecte à la base pour envoyer les modifications sur les données.
  - ⇒ Si l'application charge des données de la base, qu'elle se déconnecte de la base et qu'elle exploite les données. Dans ce cas, on dit que l'application accède à la base en lecture seule.
  - ⇒ Si l'application récupère des données à partir d'une autre source que la base de données (par exemple à partir d'une saisie manuelle d'un utilisateur ou d'un document XML) puis qu'elle se connecte à la base pour y stocker les nouvelles données.
  - ⇒ Si l'application n'utilise pas la base de données. Préciser ce cas a un sens, car une application peut travailler avec des classes de ADO.NET sans pour autant utiliser une base de données. Par exemple, on verra que les classes de ADO.NET sont particulièrement adaptées pour la présentation de données au format XML.

Avec ADO.NET, on peut travailler soit en mode déconnecté soit en mode connecté. De nombreuses fonctionnalités intéressantes sont disponibles pour chacun des deux modes.

La faiblesse du mode connecté est qu'il génère de très nombreux accès à la base de données et plus généralement, il génère de nombreux accès réseau si la base de données est séparée physiquement du reste de l'application.

La faiblesse du mode déconnecté est qu'il amène à consommer beaucoup de mémoire, puisqu'une partie de la base est récupérée en mémoire vive pour chaque appel client. Cette faiblesse peut être prohibitive pour un serveur devant gérer un grand nombre de clients, chaque client obligeant le serveur à manipuler beaucoup de données en mémoire.

I.2 LES FOURNISSEURS DE DONNEES (PROVIDERS)

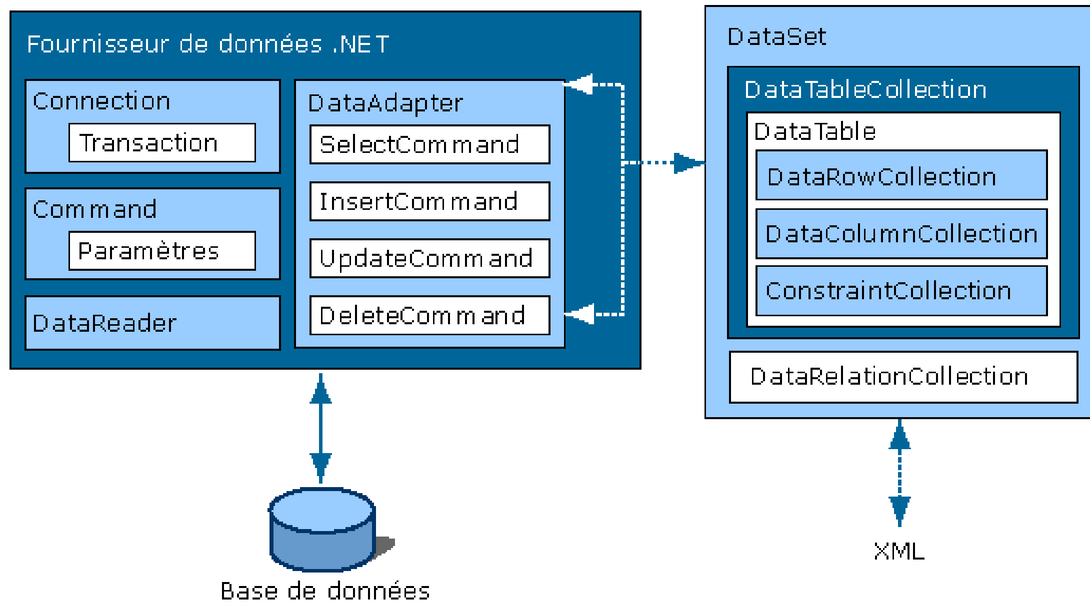
Un fournisseur de données .NET *Framework* (couche logicielle permettant de communiquer avec un SGBD relationnel spécifique), est utilisé pour la connexion à une base de données, l'exécution de commandes et l'extraction de résultats.

Voici les quatre fournisseurs de données supportés par défaut par le *framework* .NET :

- Le SGBD **SQL Server** a son propre fournisseur de données. Les classes de ce fournisseur de données se trouvent dans l'espace de noms **System.Data.SqlClient**. (à utiliser avec des versions de SQL *Serve*r 7.0, 2000 et 2005).
- Un autre fournisseur de données permet de communiquer avec les fournisseurs de données qui supportent l'API **OleDB***.* OleDB est une API permettant d'accéder aux données d'un SGBD, avec la technologie COM. Les classes de ce fournisseur de données se trouvent dans l'espace de noms **System.Data.OleDbClient**. (à utiliser avec des versions de SQL Server antérieures à 7.0.*et* Access)
- Il existe un fournisseur de données .NET qui se place au dessus du protocole **ODBC** *(Open DataBase Connectivity).* Ce fournisseur de données géré permet d'exploiter les fournisseurs de données non gérés qui supportent l'API ODBC Les classes de ce fournisseur d'accès sont disponibles dans l'espace de noms **System.Data.Odbc**.
- Il existe un fournisseur de données .NET, spécialisé pour l'utilisation de bases de données **Oracle***.* Les classes de ce fournisseur de données sont disponibles dans l'espace de noms **System.Data.OracleClient**.(à partir de la V8).

À l'exception du fournisseur de données Oracle qui se trouve dans la DLL **System.Data. OracleClient.dll** les trois autres fournisseurs de données se trouvent dans la DLL **System. Data.dll.**

I.3 SCHEMA GENERAL D'ADO.NET



Les grands principes d'ADO.NET :

- Tous les accès au SGBD se font par l'intermédiaire d'un objet **Connexion** dont l'implémentation fait partie du fournisseur de données.
- La classe **DataSet** permet de travailler en mode **déconnecté.** Dans ce mode, tout chargement ou toute sauvegarde de données se fait par l'intermédiaire d'un objet *adaptateur*.
  Un **DataSet** représente une image locale déconnectée de la base de données. Un **DataSet** est rempli avec des données provenant d'une base de données ; l'application se déconnecte, consomme les données du **DataSet**, éventuellement les modifie, puis se reconnecte à la base pour éventuellement sauver les changements effectués sur les données.
  Les données du **DataSet** sont stockées au format **XML** ; il est constitué de deux éléments :
  - ⇒ Une collection d'objets **DataTable** qui contiennent les données du DataSet. Chaque **DataTable** est constitué de plusieurs collections, notamment une collection de **DataRow** et de **DataColumn**.
  - ⇒ Une collection d'objets **DataRelation** qui définissent les contraintes sur les **DataTable** (contraintes de clés étrangères, …) qui permettent de reproduire les contraintes définies sur la base de données.
- La classe **DataReader** permet de travailler en mode **connecté**.

Chaque fournisseur de données est chargé de fournir les classes implémentant les objets suivants :

Objet	Description
Connection	Établit une connexion à une source de données spécifique. Cet objet utilise une chaîne de connexion dépendant de la base de données.
ConnectionStringBuilder	Fournit un moyen simple de créer et de gérer le contenu de chaînes de connexion utilisées par la classe **SqlConnection**.
Command	Exécute une requête SQL de tout type (extraction ou mise à jour) sur une source de données. Cet objet peut être paramétré de façon à représenter une requête générique.
DataReader	Extrait un flux de données avant uniquement (forward only) et en lecture seule à partir d'une source de données.
DataAdapter	Remplit un **DataSet** et répercute les mises à jour dans la source de données..
Exception	Exception qui est levée lorsque le fournisseur retourne un avertissement ou une erreur
Parameter	Représente un paramètre d'une **Command** et éventuellement son mappage aux colonnes **DataSet**
ParameterCollection	Représente une collection de paramètres associés à **Command** et leurs mappages respectifs à des colonnes dans **DataSet**..

Ces classes s'utilisent de la même manière, quelle que soit la base de données. De plus une convention veut que les données soient nommées selon le même modèle, quel que soit le fournisseur.

La classe qui réalise la connexion avec la base de données s'appelle sqlConnection (SQL Server), OracleConnection (Oracle) …

II LE MODELE DE PROGRAMMATION ADO.NET

Le modèle repose sur plusieurs espaces de noms :

- **System.Data**
  Comprend les classes qui constituent l'architecture ADO.NET
- **System.Data.Common**
  Fournit des classes abstraites pour la manipulation de données, qui permettent d'écrire du code pouvant fonctionner sur divers fournisseurs de données
- **System.Data.OleDb**
  Contient des classes qui constituent le fournisseur de données .NET Framework pour les sources de données compatibles OLE DB
- **System.Data.SqlClient**
  Contient des classes qui constituent le fournisseur de données .NET Framework pour SQL Server, qui permet de se connecter à SQL Server
- **System.Sql**
  Contient des classes pour des types de données natifs SQL Server
- **System.Data.ODBC**
  Contient des classes qui constituent le fournisseur de données .NET Framework pour ODBC
- **System. Data.OracleClient**
  Contient des classes qui constituent le fournisseur de données .NET Framework pour Oracle

II.1 LA CLASSE CONNEXION

Un objet générique connexion représente une session unique vers une source de données. Avec un système de base de données client/serveur, il équivaut à une connexion réseau au serveur. Lorsque vous créez une instance de connexion, les valeurs initiales sont affectées à toutes les propriétés.

Si l'objet connexion est hors de portée, il n'est pas fermé. Par conséquent, la connexion doit être fermée explicitement en appelant **Close** ou **Dispose**. Ils sont équivalents sur le plan des fonctionnalités.

Si la valeur de regroupement de connexion **Pooling** est **true** ou **yes**, la connexion physique est aussi libérée. La connexion peut également être ouverte dans un bloc **using**, ce qui garantit que la connexion soit fermée lorsque le code quitte ce bloc.

Les principales propriétés disponibles pour l’objet connexion, quel que soit le provider.

Nom	Description
ConnectionString	Obtient ou définit la chaîne permettant d'ouvrir la base de données.
ConnectionTimeout	Obtient la durée d'attente préalable à l'établissement d'une connexion avant que la tentative ne soit abandonnée et qu'une erreur ne soit générée.
Database	Obtient le nom de la base de données en cours ou de la base de données à utiliser une fois la connexion ouverte.
DataSource	Obtient le nom de l'instance à laquelle se connecter.
ServerVersion	Obtient une chaîne comportant la version du server à laquelle le client est connecté
State	Indique l'état actuel de la connexion

Les principales méthodes disponibles pour l’objet **Connection**, quel que soit le provider.

Nom	Description
BeginTransaction	Commence une transaction de base de données.
ChangeDatabase	Modifie la base de données en cours d'une Connection ouverte.
Close	Substitué. Ferme la connexion à la base de données. Il s'agit de la méthode privilégiée pour la fermeture de toute connexion ouverte.
CreateCommand	Crée et retourne un objet Command associé à **Connection**.
Dispose	Surchargé. Libère les ressources utilisées par Component. (hérité de Component.)
GetSchema	Retourne des informations de schéma pour la source de données de **Connection**.
Open	Substitué. Ouvre une connexion de base de données avec les paramètres de propriété spécifiés par **ConnectionString**.

Rappel : La classe qui réalise la connexion avec la base de données s’appelle **SqlConnection** (pour SQL Server), **OracleConnection** (pour Oracle) …

Pour se connecter à la base de données, il faut renseigner la chaîne de connexion (propriété ***ConnectionString***), qui contient les informations nécessaires pour localiser et se connecter, puis ouvrir la connexion avec la méthode ***Open()***.

Exemple : Se connecter à une base SQL Server

```
// Ajout de l'espace de nom
using System.Data.SqlClient;

// objet ADO.net
private SqlConnection sqlConnect;

// accés à la base
sqlConnect = new SqlConnection();
sqlConnect.ConnectionString = "Data Source=PCSERVER;Initial
Catalog=BDTest;Integrated Security=True";

// Ouvre la connection.
try
{
  sqlConnect.Open();
}
catch (Exception e)
{
  MessageBox.Show("Erreur de connexion à la base", "Connexion",
  MessageBoxButtons.OK, MessageBoxIcon.Information);

}
finally
{
  sqlConnect.Close();
}
```

que l'on pourrait également coder :

```
// accés à la base
string connectString ="Data Source=PCSERVER;Initial
Catalog=BDTest;Integrated Security=True";
sqlConnect = new SqlConnection(connectString);

// Ouvre la connection.
try
{
  sqlConnect.Open();
}
catch (SqlException se)
{
// ...
}
finally
{
  sqlConnect.Close();
}
```

que l'on pourrait aussi coder :

```
// accés à la base
string connectString ="Data Source=PCSERVER;Initial
Catalog=BDTest;Integrated Security=True";
using (sqlConnect = new SqlConnection(connectString))
{
  try
  {
   sqlConnect.Open();
  }
  catch (SqlException se)
  {
   // Traitement des erreurs
  }
}
```

SqlConnection est fermé automatiquement à la fin du bloc du code **using**.

Exemple de chaîne de connexion plus complète :

```
string connectString = "initial catalog=BDTest;integrated
security=SSPI;persist security info=False;workstation
id=DIPSI02;packet size=4096";
```

ou

DataSource représente le nom du serveur,
InitialCatalog représente le nom de la base de données
Integrated Security=true indique l'utilisation de l'authentification Windows intégrée

L'attribution au mot clé **Persist Security Info** de la valeur **false** garantit que des informations sensibles (ID utilisateur ou mot de passe) ne seront pas écrites sur disque.
workstation id représente le nom de l'ordinateur client
packet size indique la taille (en octets) des paquets réseau permettant de communiquer avec une instance de SQL Server.

Exemple de chaîne de connexion pour une base ACCESS : les mots clés **UserID** et **Password** sont facultatifs si la base de données n'est pas sécurisée (par défaut)

```
string connectString = "Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=c:\Northwind.mdb;User ID=Admin;Password=;";
```

Le tableau suivant énumère les noms les plus utilisés des valeurs de mot clé dans la chaîne **ConnectionString**.

Mot clé	Par défaut	Description
Application Name	N/A	Nom de l'application ou '.NET SQLClient Data Provider' si aucun nom d'application n'est fourni.
Connect Timeout - ou - **Connection Timeout**	15	Durée d'attente (en secondes) préalable à l'établissement d'une connexion au serveur avant que la tentative ne soit abandonnée et qu'une erreur ne soit générée.

Data Source
- ou -
Server
- ou -
Address
- ou -
Addr
- ou -
Network Address
	N/A	Nom ou adresse réseau de l'instance de SQL Server à laquelle se connecter. Le numéro de port peut être spécifié après le nom du serveur :
		`server=tcp:servername, portnumber`
		Lors de la spécification d'une instance locale, utilisez toujours (local). Pour forcer un protocole, ajoutez l'un des préfixes suivants :
		`np:(local), tcp:(local), lpc:(local)`

Encrypt	'false'	Si **true**, SQL Server utilise le chiffrement SSL pour tous les échanges de données se produisant entre le client et le serveur si celui-ci dispose d'un certificat installé. Les valeurs reconnues sont **true**, **false**, **yes** et **no**.
Initial Catalog - ou - **Database**	N/A	Nom de la base de données.
Integrated Security - ou - **Trusted_Connection**	'false'	Lorsque la valeur est **false**, l'ID d'utilisateur et le mot de passe sont spécifiés dans la connexion. Lorsque la valeur est **true**, les informations d'identification actuelles du compte Windows sont utilisées pour l'authentification. Les valeurs reconnues sont **true**, **false**, **yes**, **no** et **sspi** (vivement recommandée), qui équivaut à **true**.

Packet Size	8192	Taille en octets des paquets réseau permettant de communiquer avec une instance de SQL Server.
Password - ou - **Pwd**	N/A	Le mot de passe pour le compte SQL Server qui se connecte. Non recommandé. Pour garantir un niveau de sécurité élevé, il est vivement recommandé d'utiliser de préférence le mot clé **Integrated Security** ou **Trusted_Connection**.
Persist Security Info	'false'	Lorsqu'elles ont comme valeur **false** ou **no** (vivement recommandée), les informations de sécurité, comme le mot de passe, ne sont pas retournées dans le cadre de la connexion si celle-ci est ouverte ou l'a été à un moment donné. Le rétablissement de la chaîne de connexion rétablit toutes les valeurs des chaînes de connexion, y compris le mot de passe. Les valeurs reconnues sont **true**, **false**, **yes** et **no**.
TrustServerCertificate	'false'	Si la valeur définie est **true**, SSL est utilisé pour chiffrer le canal mais en ignorant l'approbation de la chaîne de certificats. Si **TrustServerCertificate** a la valeur **true** mais que **Encrypt** n'a pas la valeur **true** pour la chaîne de connexion, le canal n'est pas chiffré. Les valeurs reconnues sont **true**, **false**, **yes** et **no**. Pour plus d'informations, consultez « Encryption Hierarchy » et « Using Encryption Without Validation » dans la documentation en ligne de SQL Server 2005.
User ID	N/A	Compte de connexion SQL Server. Non recommandé. Pour garantir un niveau de sécurité élevé, il est vivement recommandé d'utiliser de préférence les mots clés **Integrated Security** ou **Trusted_Connection**.
Workstation ID	Nom de l'ordinateur local.	Nom du poste de travail en cours de connexion à SQL Server

Une chaîne de connexion sera facilement obtenue en recopiant la chaîne générée grâce à l'IDE, Menu Données / Ajouter une nouvelle source de Données, Choix d'une base et Suivant …

Elle pourra être également créée grâce à la classe **SqlConnectionStringBuilder** qui fournit un moyen simple de créer et de gérer le contenu de chaînes de connexion utilisées par la classe **SqlConnection**.

Exemple : Utilisation de la classe **SqlConnectionStringBuilder** :

```
// objet ADO.net
private SqlConnection sqlConnect;

// création de la chaîne de connection
SqlConnectionStringBuilder obuilder = new
SqlConnectionStringBuilder();
obuilder["Data Source"] = "(local)";
obuilder["integrated Security"] = true;
obuilder["Initial Catalog"] = "BDTest";

sqlConnect = new SqlConnection(obuilder.ConnectionString);
```

II.1.1 Stocker les chaînes de connexion

Par sécurité, il est recommande de ne pas incorporer de chaînes de connexion dans le code (des chaînes de connexion chiffrées compilées dans le code source d'une application peuvent être affichées à l'aide du désassembleur MSIL Disassembler (ildasm.exe))

Les chaînes de connexion peuvent être stockées dans un fichier de configuration xxx.config, situé dans le répertoire du projet.

Tout fichier de configuration .NET admet un élément racine <configuration> ; la liste ci-dessous présente les sous éléments standard de l'élément configuration.

- <appSettings> éléments de configuration propre à l'application
- <configSections> permet de définir les sections de configuration
- <connectionString> stocke les chaînes de connexion aux bases de données
- <location> permet de définir les éléments de configuration ASP.Net pour chaque page Web
- <system.Data.[Fournisseur de données]>permet de paramétrer les fournisseurs de données ADO.Net
- <system.Transaction> permet de paramétrer les transactions
- < system.Web> définit les paramètres utilisés par ASP.Nett

Exemple : Codage d'une chaine de connection dans le fichier de configuration :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <configSections>
   </configSections>
   <connectionStrings>
        <add name="BDTest"
        connectionString="Data Source=PCSERVER;Initial
        Catalog=BDTest;Integrated Security=True"
        providerName="System.Data.SqlClient" />
   </connectionStrings>
</configuration>
```

Le codage d'une chaîne de connexion dans le fichier de configuration de l'application peut être enregistré directement dans les **Propriétés** du projet, sous l'onglet **Paramètres** :

- Le champ **Nom** contient le nom de la chaîne,
- Le champ **Type** contient la valeur chaîne de connexion,
- Le champ **Portée** contient la valeur Application,
- Le champ **Valeur** contient la chaîne de connexion (saisie facilitée par le bouton de sélection).

II.1.2 Extraction des chaînes de connexions

Pour l'utiliser dans le code, il faut pouvoir l'extraire la chaîne de connexion stockée dans le fichier de configuration.

L'espace de noms **System.Configuration** fournit des classes pour utiliser les informations de configuration stockées dans des fichiers de configuration. La classe **ConnectionStringSetting**s représente une chaîne de connexion et comprend deux propriétés :

- **ConnectionString** représente la valeur de la chaîne de connexion,
- **Name** représente le nom de la chaîne de connexion.

La classe **ConfigurationManager** permet d'accéder aux informations de configuration machine et de l'application, et retourne grâce à sa propriété **ConnectionStrings**, la collection des chaînes de connexion codées dans la section <connectionStrings> du fichier de configuration.
Attention : ne pas oublier d'importer l'espace de nom **System.Configuration** dans le projet (Click droit sur **Référence / Ajouter une référence**).

Exemple : Extraction d'une chaine de connection du fichier de configuration :

```
// Ajout de l'espace de nom
using System.Data.SqlClient;
using System.Configuration;
// objet ADO.net
private SqlConnection sqlConnect;
// accés à la base
sqlConnect = new SqlConnection();
ConnectionStringSettings oConfig =
ConfigurationManager.ConnectionStrings["BDTest"];
```

```
if (oConfig != null)
{
  // affectation de la chaine de connection extraite
  sqlConnect.ConnectionString = oConfig.ConnectionString;
}
```

II.1.3 Pool de connexions

L'utilisation d'un pool de connexions peut augmenter les performances de manière significative : un pool de connexion est une collection de connexions équivalentes, connectées à la même base, réutilisables par une nouvelle demande client.

L'idée sous jacente est de recycler les connexions déjà ouvertes, de façon à diminuer les coûts de création, d'initialisation et de destruction.

La gestion en interne du pool de connexion dépend du fournisseur de données. Seules des connexions présentant la même configuration peuvent être regroupées. ADO.NET conserve plusieurs pools en parallèle, un par configuration.

Chaque fois qu'un utilisateur appelle **Open** sur une connexion, le dispositif de regroupement de connexions vérifie s'il y a une connexion disponible dans le pool. Si une connexion regroupée est disponible, le dispositif de regroupement de connexions la retourne à l'appelant au lieu d'ouvrir une nouvelle connexion. Lorsque l'application appelle **Close** sur la connexion, le dispositif de regroupement de connexions la retourne à l'ensemble regroupé de connexions actives au lieu de la fermer réellement. Une fois la connexion retournée au pool, elle est prête à être réutilisée sur l'appel de **Open** suivant.

Si **MinPoolSize** n'est pas spécifié dans la chaîne de connexion ou est spécifié comme zéro, les connexions du pool sont fermées après une période d'inactivité. Lorsqu'un pool est créé, plusieurs objets de connexion sont créés et ajoutés au pool de sorte que l'exigence de taille minimale du pool soit remplie. Les connexions sont ajoutées au pool jusqu'à ce que sa taille maximale spécifiée soit atteinte (la valeur par défaut est 100).

(voir **Utilisation du regroupement de connexions** dans l'aide de Visual Studio)

II.1.4 Obtenir des informations de schéma relationnel d'une base de données

A partir d'une connexion OLEDB sur un serveur SGBDR, il est possible d'obtenir des informations sur la structure des objets de la base de données au travers de la méthode ***GetSchema*** de l'objet connexion.

Remarques : la présence d'un schéma relationnel de base de données (catalogue de la structure) est présent sur tout SGBD conforme à la norme SQL.

II.1.5 Intercepter des évènements issus d'une connexion

Il peut être intéressant d'intercepter l'événement de changement d'état **StateChange** d'une connexion ADO, qui se produit lorsque l'événement passe de l'état fermé à l'état ouvert, ou vice versa.

II.2 LA CLASSE COMMANDE

La classe commande permet d'exécuter des requêtes SQL, ou des procédures stockées avec ou sans paramètres. Les commandes exécutées peuvent ou non retourner des résultats.
Les principales propriétés disponibles pour l'objet **Command**, quel que soit le provider.

Nom	**Description**
CommandText	Obtient ou définit l'instruction Transact-SQL ou la procédure stockée à exécuter au niveau de la source de données.
CommandTimeout	Substitué. Obtient ou définit la durée d'attente qui précède le moment où il est mis fin à une tentative d'exécution d'une commande et où une erreur est générée.
CommandType	Substitué. Obtient ou définit une valeur indiquant la manière dont la propriété *CommandText* doit être interprétée.
Connection	Obtient ou définit l'objet **Connection** utilisé par cette instance de **Command**.
Parameters	Obtient la collection des paramètres de **Commande**
Transaction	Obtient ou définit la transaction dans laquelle **Command** s'exécute.
UpdatedRowSource	Obtient ou définit la manière dont les résultats des commandes sont appliqués à *DataRow* lorsqu'ils sont utilisés par la méthode **Update** de *DbDataAdapter*.

Les principales méthodes disponibles pour l'objet **Command**, quel que soit le provider.

Nom	**Description**
Cancel	Tente d'annuler l'exécution de l'objet **Command**.
CreateParameter	Crée une nouvelle instance d'un objet Parameter.
ExecuteNonQuery	Exécute une instruction Transact-SQL sur la connexion et retourne le nombre de lignes affectées.
ExecuteReader	Envoie **CommandText** à **Connection** et génère **DataReader**.
ExecuteScalar	Substitué. Exécute la requête et retourne la première colonne de la première ligne du jeu de résultats retourné par la requête. Les colonnes ou lignes supplémentaires sont ignorées.
ExecuteXmlReader	Envoie **CommandText** à **Connection** et génère un objet **XmlReader**.
Prepare	Substitué. Crée une version préparée de la commande sur une instance du server.

Rappel : La classe **Command** s'appelle **SqlCommand** (pour SQL Server), OracleCommand (pour Oracle) …

Seul le fournisseur **SqlClient** permet une exécution asynchrone, dont les principales méthodes sont listées ci-dessous.

Nom	**Description**
BeginExecuteNonQuery	Lance l'exécution asynchrone de l'instruction Transact-SQL ou de la procédure stockée qui est décrite par SqlCommand.
EndExecuteNonQuery	Termine l'exécution asynchrone d'une instruction Transact-SQL.
BeginExecuteReader	Lance l'exécution asynchrone de l'instruction Transact-SQL ou de la procédure stockée qui est décrite par **SqlCommand** et récupère un ou plusieurs jeux de résultats du serveur.
EndExecuteReader	Termine l'exécution asynchrone d'une instruction Transact-SQL en retournant le DataReader demandé.

Les objets de type commande peuvent être **créés** de 3 façons :

1. Créer une nouvelle instance de commande avec le mot clé New et en définir les propriétés, en particulier les propriétés **Connection** qui associera la commande à la connection établie, **CommandText** qui contiendra la chaîne de requête SQL ou le nom de la procédure stockée, et la propriété **CommandType** qui indiquera comment doit être interprétée la propriété **CommandText**.
2. Utiliser un constructeur en lui indiquant la chaîne de requête à exécuter et l'objet connexion.
3. Faire appel à la méthode **CreateCommand** de l'objet connexion.

Les objets de type commande peuvent être **utilisés** de 3 façons :

1. Avec un **DataReader** pour récupérer un lot de lignes de données grâce à la méthode **ExecuteReader** de l'objet **Commande**.
2. Pour obtenir une valeur calculée à partir des informations contenues dans la base de données (à partir de fonctions SQL Server telles Count, Min, Max, Sum …) grâce à la méthode **ExecuteScalar** de l'objet **Commande**.
3. Pour exécuter des requêtes de modification, ajout et suppression de données grâce à la méthode **ExecuteNonQuery** de l'objet **Commande**..

II.2.1 Obtenir des données avec l'objet DataReader

L'objet générique **DataReader** offre un accès connecté rapide et léger pour lire des lignes d'une source de données (curseur unidirectionnel, en avant seulement, en lecture seule).

Le **DataReader** est limité, mais hautement optimisé. Il est important de connaître ses caractéristiques avant de décider quel mécanisme utiliser dans une application pour accéder aux données :

- Pas de besoin de réaliser un cache des données
- Le jeu d'enregistrement est trop important pour être stocké en mémoire
- Accès rapide aux données en lecture seule en avant seulement.
- Accès aux données en mode connecté.

Les principales propriétés disponibles pour l'objet **DataReader**.

Nom	Description
FieldCount	Obtient le nombre de colonnes figurant dans la ligne actuelle.
HasRows	Obtient une valeur qui indique si ce **DataReader** contient une ou plusieurs lignes.
IsClosed	Obtient une valeur indiquant si **DataReader** est fermé.
Item	Surchargé. Obtient la valeur d'une colonne spécifiée sous la forme d'une instance de Object.
RecordsAffected	Obtient une valeur indiquant si **DataReader** contient une ou plusieurs lignes.

Les principales méthodes disponibles pour l'objet **DataReader**.

Nom	Description
Close	Ferme l'objet **DataReader**.
GetBoolean	Obtient la valeur de la colonne spécifiée sous la forme d'une valeur Boolean.
GetByte	Obtient la valeur de la colonne spécifiée sous la forme d'un octet.
GetChar	Obtient la valeur de la colonne spécifiée sous la forme d'un caractère unique.
GetDataTypeName	Obtient le nom du type de données de la colonne spécifiée.
GetDateTime	Obtient la valeur de la colonne spécifiée sous la forme d'un objet DateTime.
GetDecimal	Obtient la valeur de la colonne spécifiée sous la forme d'un objet Decimal.
GetDouble	Obtient la valeur de la colonne spécifiée sous la forme d'un

	nombre à virgule flottante double précision.
GetFieldType	Obtient le type de données de la colonne spécifiée.
GetFloat	Obtient la valeur de la colonne spécifiée sous la forme d'un nombre à virgule flottante simple précision.
GetGuid	Obtient la valeur de la colonne spécifiée sous la forme d'un identificateur global unique (GUID, Globally Unique IDentifier).
GetInt16	Obtient la valeur de la colonne spécifiée sous la forme d'un entier signé 16 bits.
GetInt32	Obtient la valeur de la colonne spécifiée sous la forme d'un entier signé 32 bits.
GetInt64	Obtient la valeur de la colonne spécifiée sous la forme d'un entier signé 64 bits.
GetName	Obtient le nom de la colonne, en fonction de l'ordinal de colonne de base zéro.
GetOrdinal	Obtient l'ordinal de colonne, en fonction du nom de la colonne.
GetSchemaTable	Retourne un **DataTable** qui décrit les métadonnées de colonne de **DataReader**.
GetString	Obtient la valeur de la colonne spécifiée sous la forme d'une instance de String.
GetType	Obtient le Type de l'instance en cours. (hérité de Object.)
GetValue	Obtient la valeur de la colonne spécifiée sous la forme d'une instance de **Object**.
GetValues	Obtient toutes les colonnes d'attributs figurant dans la collection de la ligne actuelle.
IsDBNull	Obtient une valeur qui indique si la colonne contient des valeurs inexistantes ou manquantes.
Read	Avance le lecteur à l'enregistrement suivant dans un jeu de résultats.

La lecture d'une ligne de données extraite de la base sera effectuée par la méthode **Read** ; chaque colonne de la ligne sera restituée à l'application par une méthode **Getxxx**, selon le type de donnée contenu dans la colonne.
Le DataReader sera ensuite fermé par la méthode **Close()**.

Rappel : L'objet DataReader générique est instance de la classe **SqlDataReader** (pour SQL Server), **OracleDataReader** (pour Oracle) …

Exemple 1: Récupérer le code et le nom de tous les fournisseurs, à partir de la table FOURNIS de la base SQL Server de nom BDTest

Dont les 2 premiers champs sont :

NUMFOU : N° de compte fournisseur de type int

NOMFOU : Nom fournisseur de type varchar(30)

```
// objets ADO.net
private System.Data.SqlClient.SqlConnection sqlConnect;
private System.Data.SqlClient.SqlCommand sqlCde;
private System.Data.SqlClient.SqlDataReader sqlRdr;
// création de la connection
sqlConnect = new SqlConnection();
sqlConnect.ConnectionString = "Data Source=PCSERVER;Initial
Catalog=BDTest;Integrated Security=True";
try
{
  // Ouvre la connection.
  sqlConnect.Open();
  // Création de la commande
  sqlCde = new SqlCommand();❶
  sqlCde.Connection = sqlConnect;❷
  // Constitution Requête SQL
  string strSql ="Select * from Fournis";❸
  // Positionnement des propriétés
  sqlCde.CommandType=CommandType.Text;❹
  sqlCde.CommandText = strSql;❺

  // Exécution de la commande
  sqlRdr=sqlCde.ExecuteReader();❻

  // Lecture des données du DataReader
  while (sqlRdr.Read())❼
  {
   int codeF = sqlRdr.GetInt16(0); ❽
   string nomF = sqlRdr.GetString(1);
   …
  }
  // Fin des données
  sqlRdr.Close() ;❾
}
catch (SqlException se)
{
  // Traitement des erreurs
}

finally
{
 sqlConnect.Close();❿
}
```

❶ La commande est créée suivant la méthode proposée au point 1 des méthodes de création d'une commande.

❷ Définition de la connexion utilisée par cette commande

❸ Codification de la requête SQL dans une variable de type chaîne.

❹ La propriété **CommandText** sera interprétée comme une chaîne SQL (valeur par défaut).

❺ Affectation de la propriété **CommandText** avec la chaîne de requête définie.

❻ Exécution de la requête
Le lot de lignes est constitué par l'appel de la méthode **ExecuteReader** de l'objet de classe **SqlCommand**.

❼ Les lignes de données sont lues par une boucle while.
Les lignes sont obtenues de la base, une à une par l'appel de la méthode **Read()** de l'objet de classe **SqlDataReader** , qui renvoie un booléen positionné à *false* en fin de lot.

❽ Il est possible d'accéder aux valeurs des colonnes soit par le numéro de la colonne de la requête SQL, soit par le nom de la colonne.

Exemple :

```
int Code = (int)sqlRdr[0] ;
```

ou

```
int Code = sqlRdr["Nomfou"].ToString();
```

Une solution plus performante est proposée permettant d'accéder aux valeurs dans leurs types de données natifs (GetInt32, GetDouble, GetString, …).

```
int Code = sqlRdr.GetInt16(0);
```

renvoie la valeur de la colonne 0 sous la forme d'un entier signé 16 bits.

```
sqlRdr. GetString(1)
```

renvoie la valeur de la colonne 1 sous la forme d'une chaîne.

❾ La méthode Close doit être appelée en fin de lecture des données ; elle remplit les valeurs des paramètres de sortie, les valeurs de retour et la propriété **RecordsAffected.** Notez que pendant l'ouverture d'un **DataReader**, **Connection** est utilisé en mode exclusif par ce **DataReader**. Vous ne pourrez pas exécuter les commandes pour **Connection**, y compris la création d'un autre **DataReader**, jusqu'à la fermeture du **DataReader** d'origine.

❿ Fermeture de la connexion.
Le bloc **try…finally** sécurise le code : erreur ou pas erreur à l'intérieur du bloc, la connexion sera fermée

On utilisera la méthode IsDbNull() pour tester le retour de la base des champs Null.

```
if (!(sqlRdr.IsDBNull(2))) txtR.Text = sqlRdr.GetString(2).Trim();
```

II.2.2 Obtenir une valeur de la base de données

En utilisant le langage SQL, directement au niveau du SGBD, on peut obtenir des valeurs simples à partir des informations contenues dans la base :
Type d'utilisation : compter un nombre de lignes d'une table : il serait trop coûteux de ramener la table entière dans un **DataReader** pour compter le nombre de lignes …

Exemple 2: Obtenir le nombre de lignes de la table Fournis de la base SQL Server BDTest.

```
// objets ADO.net
private System.Data.SqlClient.SqlConnection sqlConnect;
private System.Data.SqlClient.SqlCommand sqlCde;
// accés à la base
sqlConnect = new SqlConnection();
sqlConnect.ConnectionString = "Data Source=PCSERVER;Initial
Catalog=BDTest;Integrated Security=True";
try
{
  // Ouvre la connection.
  sqlConnect.Open();
  // Création de la commande
  string strSql = "Select Count(*) From Fournis";
  sqlCde = new SqlCommand(strSql, sqlConnect); ❶
  int nbPays=(int)sqlCde.ExecuteScalar() ;❷
}
catch (SqlException se){// Traitement des erreurs }
finally {sqlConnect.Close();}
```

❶ La commande est créée suivant la méthode proposée au point 2 des méthodes de création d'une commande, en utilisant un constructeur en lui indiquant la chaîne de requête à exécuter et l'objet connexion.

❷ Le nombre de pays a été ramené de la base grâce à l'exécution de la méthode **ExecuteScalar** de l'objet de classe **SqlCommand** ; La méthode retourne un type objet : c'est la raison pour laquelle le cast est nécessaire.

De manière générale, la méthode **ExecuteScala**r retourne la première colonne de la première ligne du jeu de résultats retourné par la requête. Toutes les autres colonnes et lignes sont ignorées.

II.2.3 Mettre à jour les données avec des requêtes SQL

En utilisant les requêtes SQL INSERT, UPDATE ou DELETE, on va pouvoir mettre à jour les données de la base, par l'intermédiaire de l'objet **Commande** et de sa méthode **ExecuteNonQuery(),** qui retourne le nombre de lignes traitées par la requête, ce qui peut permettre de contrôler la bonne exécution de l'opération.

Exemple 3: Supprimer de la table Produit l'article de code 'I100'.

```
// objets ADO.net
private System.Data.SqlClient.SqlConnection sqlConnect;
private System.Data.SqlClient.SqlCommand sqlCde;
// accés à la base
sqlConnect = new SqlConnection();
sqlConnect.ConnectionString = "Data Source=PCSERVER;Initial
Catalog=BDTest;Integrated Security=True";

try
{
  // Ouvre la connection.
  sqlConnect.Open();
  // Constitution Requête SQL et exécution de la commande
  strSql = "Delete Produit where codart = 'I100'" ; ❶
  // Création de la commande
  sqlCde = new SqlCommand(strSql, sqlConnect);
  int n = sqlCde.ExecuteNonQuery();❷
  // retourne le nombre de lignes affectées
  if (n == 1)
  {
    MessageBox.Show("Suppression effectuée", "Confirmation de
    suppression");
  }
  else
  {
    MessageBox.Show("La suppression a échoué : Veuillez
    recommencer", "Confimation de suppression");}
 }
catch (SqlException se){// Traitement des erreurs }
finally {sqlConnect.Close();}
```

❶ Préparation de la chaîne de requête de mise à jour de données

❷ Exécution de la requête, et retour du nombr de lignes affectées. On peut imaginer que si nest différent de 1, il est égal à 0 : le cas pourrait se produire dans un environnement multiutisateur, ou 2 utilisateurs ont souhaité effectué une suppression du même article au même moment ;pour le premier utilisateur, la suppression est effectuée, pour le deuxième, la suppression ne peut pas aboutir puisque la ligne n'existe plus.

II.2.4 Utiliser des procédures stockées

Dans tous ces exemples, on a utilisé une requête SQL, directement codée dans le source de l'application.

On aurait pu aussi utiliser une procédure stockée, créée préalablement dans la base.

La proximité d'un traitement dans une procédure stockée avec les données elles-mêmes est un avantage, mais complexifie en général la maintenance globale de l'application, notamment parce que plusieurs langages de programmation sont utilisés.

L'exemple 1, récupérer les lignes de la table Fournis de la base SQL Server BDTest, se transformerait ainsi :

```
…
// Création de la commande
sqlCde = new SqlCommand();
sqlCde.Connection = sqlConnect;❶

// Paramétrage de la commande
sqlCde.CommandType=CommandType.StoredProcedure;❷
sqlCde.CommandText="GetFournis";❸

// exécution de la commande
sqlRdr=sqlCde.ExecuteReader();
…
```

❶ Création de la commande.

❷ La propriété **CommandText** sera interprétée comme une procédure stockée.

❸ Affectation de la propriété **CommandText** avec le nom de la procédure stockée.

II.2.5 Paramétrer requêtes et procédures stockées

L'exemple 1, récupérer les lignes de la table Produit à partir d'une base SQL Server, se transformerait ainsi :

Dans un paragraphe précédent (Exemple 3), nous avons supprimé une ligne de la table Produit de code fourni *« en dur »* dans la requête SQL. Nous pouvons aisément imaginer que le code produit puisse être paramétrable, et donc saisi, par exemple dans une TextBox.

La valeur saisie doit donc être passée en paramètre à la requête SQL par une concaténation de chaîne, ou alors être passée en paramètre d'une procédure stockée.

En utilisant une concaténation de chaînes

Exemple 1: Dans la table Produit, sélectionner toutes les colonnes de l'article de code entré en paramètre dans la TextBox de nom txtCodeP:

```
// constitution Requête SQL et exécution de la commande
strSql ="Select * from Produit where Codart =' " + txtCodeP.Text
+ " ' ";
```

ou

```
strSql = string.Format("Select * from Produit where Codart ='{0}'",
txtCodeP.Text);
```

Attention : Notez bien l'encadrement de la valeur saisie en paramètre par des quottes, signifiant que le type de données reçu est alpha …attention aux valeurs saisies avec des apostrophes qu'il faudra dédoubler pour que la requête SQL soit valide !
Lors de concaténation de champ de type numérique, les quottes sont absentes.

Exemple 2 : Dans la table Fournis, sélectionner toutes les colonnes du fournisseur de code entré en paramètre dans la TextBox de nom txtCodeF :

```
// constitution Requête SQL et exécution de la commande
strSql ="Select * from Fournis where codart = " + txtCodeF.Text ;
```

ou

```
strSql = string.Format("Select * from Fournis where numfou ={0}",
txtCodeF.Text);
```

En paramétrant la procédure stockée

La classe **Parameter** permet de définir chaque paramètre associé à une commande via la collection **Parameters**.

Les objets de type paramètre peuvent être créés en utilisant :

1. Un des sept constructeurs disponibles
2. La méthode Add de la collection Parameters Collection (7 surcharges)
3. La méthode CreateParameter de l'objet Command.

Les principales propriétés disponibles pour l'objet **Parameter**.

Nom	**Description**
DbType	Obtient ou définit le type de données du paramètre.
Direction	Obtient ou définit une valeur qui indique si le paramètre est un paramètre d'entrée uniquement, de sortie uniquement, bidirectionnel ou de valeur de retour d'une procédure stockée.
IsNullable	Obtient ou définit une valeur qui indique si le paramètre accepte les valeurs null.
ParameterName	Substitué. Obtient ou définit le nom de SqlParameter.
Value	Substitué. Obtient ou définit la valeur du paramètre.

Exemple 3: Supprimer de la table Produit, l'article de code entré en paramètre dans la TextBox de nom txtCodeP, en utilisant un paramètre

La procédure stockée est définie ainsi dans la base

```
CREATE PROCEDURE DelProduit
(@pcodart char(4))
AS
delete produit where codart=@pcodart
```

Notez que le nom des paramètres (ici `(@pcodart`) devront correspondre au nom des paramètres définis dans le code.

Le paramétrage de la commande associée pourrait se présenter sous la forme :

```
…
// Paramétrage de la commande
sqlCde.CommandType=CommandType.StoredProcedure;
sqlCde.CommandText="DelProduit";
SqlParameter p1 = new SqlParameter("@pcodart",SqlDbType.Char);
p1.IsNullable = false;
p1.Direction = ParameterDirection.Input;
p1.Value = txtCodeP.Text;
sqlCde.Parameters.Add(p1);
…
```

ou sous la forme

```
…
// Paramétrage de la commande
sqlCde.CommandType=CommandType.StoredProcedure;
sqlCde.CommandText="DelProduit";
sqlCde.Parameters.Add(new System.Data.SqlClient.SqlParameter
("@pcodart", System.Data.SqlDbType.Char)).Value=txtCodeP.Text;
…
```

L'appel d'une procédure stockée définissant un paramètre de sortie pourra être codé ainsi :

```
// Paramétrage de la commande
sqlCde.CommandType=CommandType.StoredProcedure;
sqlCde.CommandText="XXXX";
SqlParameter p1 = sqlCde.Parameters.Add ("@p1Out",SqlDbType.Int);
p1.Direction=ParameterDirection.Output;
// Exécution de la commande
sqlCde.ExecuteNonQuery();
// récup paramètre de sortie
int nb =(int)sqlCde.Parameters["@p1Out "].Value;
```

On pourrait également obtenir la valeur du paramètre en utilisant sa position dans la collection, sachant que la position 0 de la collection **Parameters** contient toujours la valeur du code retour de la procédure stockée. Le premier paramètre se trouvera donc en position 1.
On pourra donc écrire :

```
int nb =(int)sqlCde.Parameters[1].Value;
```

II.3 TRAVAILLER AVEC DES TRANSACTIONS LOCALES

Une transaction locale implique uniquement la base de données auquelle l'application est connectée.

Une transaction est démarrée, plusieurs commandes sont exécutées, l'opération réussit ou échoue. La transaction est alors annulée ou validée.

Cette approche est fonctionnellement similaire à l'exécution d'une procédure stockée qui regroupe plusieurs instructions.

L'utilisation du code ADO.Net permet une plus grande souplesse, mais ne change pas l'effet final.

II.3.1 Gérer les transactions locales dans ADO.Net 1.X et ADO 2.0

Une nouvelle transaction est démarrée à l'aide de la méthode **BeginTransaction** de la classe de connexion. A cette transaction peut être attribuée un nom et un niveau d'isolation.
La méthode correspond à l'implémentation SQL Server de BEGIN TRANSACTION.

Exemple :

```
SqlTransaction sqlTran =sqlConnect.BeginTransaction();❶
// Création de la commande 1
string strSql1 ="Insert ....";
SqlCommand sqlCde1 = new SqlCommand(strSql1, sqlConnect,
sqlTran); ❷
// Création de la commande 2
string strSql2 ="Update ....";
SqlCommand sqlCde2 = new SqlCommand(strSql2, sqlConnect)
sqlCde2.Transaction= sqlTran; ❸
try
{
 sqlCde1.ExecuteNonQuery();❹
 sqlCde2.ExecuteNonQuery();
 sqlTran.Commit();❺
}
catch (SqlException se)
{
 sqlTran.Rollback();❻
}
finally
{
 sqlConnect.Close();
}
```

❶ Association de l'objet de transaction à la connexion active.

❷ Ajout d'une première commande à la transaction.

❸ Ajout d'une deuxième commande.

❹ Exécution des 2 commandes.

❺ Validation de la transaction par la méthode Commit()

❻Annulation de la transaction en cas d'erreur

La transaction peut être créée en spécifiant son nom, et son niveau d'isolation.

```
SqlTransaction sqlTran =sqlConnect.BeginTransaction("Tran Vente-
stock", IsolationLevel.ReadCommitted);
```

Le niveau d'isolation d'une transaction détermine le niveau d'accès que les autres transactions ont sur les données en cours de modification avant la fin d'une transaction.

ReadUncommited :
N'active pas les verrouillages et améliore les performances, mais autorise toute transaction à lire les lignes modifiées avant qu'elle ne soit validée ou annulée (mauvaise lecture).

ReadCommited (valeur par défaut) :
Verrouille une ligne modifiée par une transaction en cours, et empêche toute autre transaction de lire cette ligne.

RepeatableRead :
Une transaction ne peut pas lire des données modifiées mais pas encore validées par une autre transaction, et ne peut modifier les données lues par la transaction active tant que celle-ci n'est pas terminée.

Serializable :
Une transaction ne peut pas lire des données modifiées mais pas encore validées par une autre transaction, et ne peut modifier les données lues par la transaction active tant que celle-ci n'est pas terminée.

Une transaction ne peut pas insérer de nouvelles lignes avec des valeurs de clés comprises dans le groupe de clés lues par des instructions de la transaction active, tant que celle-ci n'est pas terminée

Le niveau le plus élevé, **Serializable** fournit un degré élevé de protection contre les transactions concurrentes, mais requiert l'achèvement de chaque transaction avant qu'une autre ne démarre.

II.3.2 Gérer les transactions locales dans ADO 2.0

Avec les transactions gérées façon ADO 1.X, on s'aperçoit que la transaction est limitée à une seule base de données : les transactions distribuées sont gérées par des composants spécifiques.

Dans ADO.Net 2.0, les transactions locales et distribuées sont gérées par le biais d'une nouvelle classe, la classe **TransactionScope** de l'espace de noms **System.Transactions**.

Exemple : Exécution de 2 commandes sur deux bases de données différentes

```
using (TransactionScope ts = new TransactionScope())
{
  bool errTs;
  // Connection sur la première base
  string connectString ="Data Source=PCSERVER;Initial
  Catalog=BDTest1;Integrated Security=True";
  using (sqlConnect = new SqlConnection(connectString))
  {
   // Création de la commande
   string strSql ="Insert ....";
   SqlCommand sqlCde = new SqlCommand(strSql, sqlConnect);
   try
   {
    sqlConnect.Open();
    sqlCde.ExecuteNonQuery();
   }
   catch (SqlException se)
   {
    // Transaction en erreur
    errTs = true;
   }
  }
  // Connection sur la deuxième base
  string connectString ="Data Source=PCSERVER;Initial
  Catalog=BDTest2;Integrated Security=True";
  using (sqlConnect = new SqlConnection(connectString))
  {
   // Création de la commande
   string strSql ="Update ....";
   SqlCommand sqlCde = new SqlCommand(strSql, sqlConnect);
   try
   {
    sqlConnect.Open();
    sqlCde.ExecuteNonQuery();
   }
   catch (SqlException se)
   {
    errTs &= true;
   }
  }
  if(!errTs) ts.Complete();
}
```

Si une erreur se produit sur la deuxième base, toutes les modifications entrées dans la première base sont automatiquement annulées.

Lorsque la méthode Complete est invoquée, elle valide toutes les opérations effectuées dans les deux bases.

II.3.3 La gestion des erreurs

Lors d'une erreur rencontrée par le serveur (clé en double, contrainte non respectée …), un objet générique Exception est généré par le fournisseur de données .Net.

Rappel : La classe générée s'appelle **SqlException** (pour SQL Server), **OracleException** (pour Oracle) …

Les propriétés principales de cette classe sont :

Nom	Description
Class	Obtient le niveau de gravité de l'erreur retournée par le fournisseur de données .NET Framework.
Data	Obtient une collection de paires clé/valeur qui fournissent des informations supplémentaires, définies par l'utilisateur, sur l'exception.(hérité de Exception.)
Errors	Obtient une collection d'un ou plusieurs objets qui donnent des informations détaillées sur les exceptions générées par le fournisseur de données .NET Framework.
LineNumber	Obtient le numéro de la ligne qui a généré l'erreur dans le lot d'instructions Transact-SQL ou dans la procédure stockée.
Message	Obtient un message décrivant l'exception en cours.(hérité de Exception.)
Number	Obtient un numéro qui identifie le type d'erreur.
Server	Obtient le nom de l'ordinateur exécutant une instance de SQL Server qui a généré l'erreur.
Source	Substitué. Obtient le nom du fournisseur qui a généré l'erreur.
StackTrace	Obtient une représentation sous forme de chaîne des frames de la pile des appels au moment où l'exception en cours a été levée.(hérité de Exception.)
State	Obtient à partir de SQL Server un code d'erreur numérique qui représente un message d'erreur, d'avertissement ou de type "Aucune donnée trouvée".

Le numéro du message d'erreur ainsi que son texte correspondant à une entrée de la table **master.dbo.sysmessages**. sont donnés par les propriété **Number** et **Message.**.

Il est donc possible de différencier le traitement de l'erreur en fonction de son code, par exemple, afficher un message compréhensible par l'utilisateur :

Exemple :

```
switch (se.Number)
{
 case 547:
   MessageBox.Show("La suppression a échoué !" + "\n" +
   "Il existe des lignes de données dépendantes",
   "Suppression");
   break;
 default:
   MessageBox.Show("Erreur sur la base" + se.Message);
   break;
}
```

L'erreur 547 est détectée, et traitée spécifiquement: toute autre erreur sera traitée par le paragraphe `default:`

III ANNEXES

Procédure stockée avec récupération de clé générée

Soit la table tableTEST

Nom colonne	Type	
clé	int	PK
nom	nvarchar(50)	

La clé est de type compteur auto incrémentée.

Procédure stockée :

La procédure accepte comme paramètre

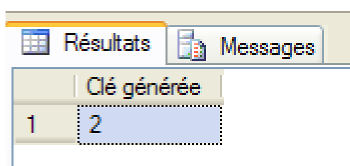
- @nom : colonne de la table
- @clegen , en sortie dans lequel sera affectée la clé générée, récupérée par la fonction@@Identity

```
Create PROCEDURE [dbo].[psAjoute]
(@pnom char(50),
 @clegen int OUTPUT)

as
-- insertion dans la table
insert into tableTest (nom) values(@pnom)

-- récupération clé générée
SET @clegen =@@IDENTITY
```

Exécution sous Studio Management

```
Declare @cle int
Exec psAjoute "Toto", @cle OUTPUT
Select @cle as "Clé générée"
```

Résultat de la requête



En C#

```
// Création commande
SqlCommand sqlCde = new SqlCommand();
sqlCde.CommandType = CommandType.StoredProcedure;
sqlCde.Connection = sqlConnect;
sqlCde.CommandText = "psAjoute";
// Spécification des parametres (diverses méthodes)
//----------------------------------------------------------------
// Parametre 1
sqlCde.Parameters.Add (new System.Data.SqlClient.SqlParameter
("@pNom",SqlDbType.Char,50)).Value =txtNom.Text;
// Parametre 2
SqlParameter pOut = new SqlParameter("@clegen", SqlDbType.Int);
pOut.Direction=ParameterDirection.Output;
SqlCde.Parameters.Add(pOut);
// Exécution de la commande
int n =sqlCde.ExecuteNonQuery();

if (n == 1)
{
  MessageBox.Show("Opération terminée avec succès. \n La clé générée
  est " + pOut.Value);
}
else
{
  MessageBox.Show("L'opération n'a pas été réalisée");
}
```

Etablissement référent
Marseille Saint Jérôme

Equipe de conception
Elisabeth Cattanéo

Remerciements :

Reproduction interdite

Article L 122-4 du code de la propriété intellectuelle.
« toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droits ou ayants cause est illicite. Il en est de même pour la traduction, l'adaptation ou la reproduction par un art ou un procédé quelconques. »

Date de mise à jour 05/05/2008
afpa © Date de dépôt légal mai 08

afpa / Direction de l'Ingénierie13 place du Générale de Gaulle / 93108 Montreuil Cedex
association nationale pour la formation professionnelle des adultes
Ministère des Affaires sociales du Travail et de la Solidarité