Utiliser ADO.NET en mode déconnecté

Dans un mode déconnecté, la connexion au serveur de base de données n'est pas permanente. Ceci permet de libérer des ressources qui pourraient être utilisées par une autre application ou par un autre client de la base de données.

Ce mode de fonctionnement implique qu'il faut conserver une copie locale des données sur lesquelles on souhaite travailler. Pour ceci, il est possible de recréer localement une structure similaire à celle d'une base de données.

1. DataSet et DataTable

Les classes principales permettant de travailler en mode déconnecté sont `DataSet`, `DataTable`, `DataRow` et `DataColumn`.

a. Description

Les différentes classes utilisées pour le mode déconnecté sont les contreparties .NET des éléments de la structure d'une base de données.

DataSet

Un objet `DataSet` est un conteneur, semblable logiquement à une base de données. C'est lui qui contiendra la totalité des données de l'ensemble sur lequel on souhaite travailler.

DataTable

Le type `DataTable` représente, comme son nom l'indique, une table. Un objet `DataTable`, bien que pouvant exister indépendamment, est logiquement contenu dans un `DataSet`.

DataColumn

La classe `DataColumn` est le pendant .NET des colonnes SQL. Les `DataTable` possèdent plusieurs `DataColumn` tout comme les tables en base de données possèdent plusieurs colonnes.

DataRow

Les objets `DataRow` sont les véritables conteneurs de données. Chaque `DataRow` correspond à un enregistrement dans une `DataTable`.

Ces quatre classes ne sont pas les seuls types représentant des éléments de la structure d'une base de données. On peut aussi trouver les types `UniqueConstraint`, `ForeignKeyConstraint` ou `DataRelation` représentant respectivement les contraintes d'unicité, les clés étrangères et les relations entre objets `DataTable`.

b. Remplissage d'un DataSet à partir d'une base de données

Pour travailler avec les données localement, il est nécessaire de rapatrier les données du serveur vers un `DataSet`. Pour cela, chaque fournisseur de données propose un `DbDataAdapter` permettant des échanges bidirectionnels entre la base de données et le `DataSet`. L'implémentation de la classe `DbDataAdapter` du fournisseur de données SQL Server est `SqlDataAdapter`.

Utilisation d'un DbDataAdapter

En premier lieu, il faut instancier un objet d'un type dérivé de la classe `DbDataAdapter`.

```
SqlDataAdapter adapteur = new SqlDataAdapter();
```

Cet objet possède une propriété `SelectCommand` à laquelle il faut affecter une commande de sélection de données. C'est à partir des données renvoyées par cette commande que l'objet `SqlDataAdapter` pourra remplir un `DataSet`.

```
SqlCommand commande = connexion.CreateCommand();
commande.CommandText = "SELECT * FROM Eleve";
adapteur.SelectCommand = commande;
```

À ce stade, il ne reste plus qu'à remplir un `DataSet` à l'aide de la méthode `Fill` du `SqlDataAdapter`. Cette méthode prend en paramètre (dans ses versions les plus simples) un `DataSet` ou une `DataTable`. Il faut donc instancier un `DataSet` et le passer à notre méthode.

```
DataSet ds = new DataSet();
adapteur.Fill(ds);
```

Il est possible de connaître le nombre de lignes ajoutées ou actualisées dans le `DataSet` en récupérant la valeur de retour de l'appel à la méthode `Fill`.

```
int nbLignes = adapteur.Fill(ds);
```

Le premier appel à la méthode `Fill` crée un élément dans la collection `ds.Tables`. Cet élément est de type `DataTable` et contient tous les enregistrements retournés par la commande de sélection. Le nom de cet objet `DataTable` (propriété `TableName`) est "Table", ce qui n'est pas particulièrement explicite. Chaque nouvelle table créée de cette manière par la méthode `Fill` aura pour nom *Table*, *Table1*, *Table2*, etc. Pour éviter ces nommages peu clairs, on utilise la méthode `Fill` en lui passant le nom de la `DataTable` à remplir :

```
adapteur.Fill(ds, "Eleve");
```

La connexion à partir de laquelle on crée la commande peut être ouverte ou fermée, dans les deux cas, tout se passera bien. La différence est dans le comportement vis-à-vis de cette connexion :

- Si la connexion n'est pas ouverte au moment de l'exécution, la méthode l'ouvrira, remplira le `DataSet` puis la refermera.
- Si la connexion est préalablement ouverte, l'exécution de la méthode peuplera uniquement le `DataSet`, sans modifier l'état de la connexion.

Lorsque le `SqlDataAdapter` construit une `DataTable`, il utilise les noms des champs retournés pour nommer chacune de ses `DataColumn`. Pour redéfinir les noms de ces champs, il est possible de créer des objets de type

DataTableMapping puis de les ajouter à la collection TableMappings du SqlDataAdapter.

La classe DataTableMapping fournit les propriétés SourceTable et DataSetTable qui permettent de mapper un nom de table en base de données à un nom de DataTable. Ce type contient également une collection ColumnMappings qu'il convient de remplir avec des objets de type DataColumnMapping dont les propriétés SourceColumn et DataSetColumn permettent respectivement de spécifier le nom du champ en base de données et le nom du champ associé dans la DataTable.

Pour appliquer ces informations de mappage, il faut appeler la méthode Fill en passant comme second paramètre le nom de la table en base de données.

Le code suivant crée un mappage pour les colonnes *Identifiant* et *DateNaissance* et remplit un objet DataTable nommé *DataTableEleve* avec les données de la table *Eleve*. Il affiche ensuite à l'écran le nom des colonnes de la DataTable.

```
var connexion = new
SqlConnection("Server=(LocalDb)\\v11.0;Initial
Catalog=Test;Integrated Security=True");
SqlCommand commande = connexion.CreateCommand();
commande.CommandText = "SELECT * From Eleve;";

SqlDataAdapter adapteur = new SqlDataAdapter();
adapteur.SelectCommand = commande;

DataTableMapping mappageTable = new DataTableMapping("Eleve",
"DataTableEleve");
mappageTable.ColumnMappings.Add(new
DataColumnMapping("Identifiant", "Id"));
mappageTable.ColumnMappings.Add(new
DataColumnMapping("DateNaissance", "Date"));
adapteur.TableMappings.Add(mappageTable);

DataSet ds = new DataSet();
adapteur.Fill(ds, "Eleve");

foreach (DataColumn column in
ds.Tables["DataTableEleve"].Columns)
{
    Console.WriteLine(column.ColumnName);
}
```

Retrouver les contraintes de la base de données dans un DataSet

L'utilisation de la méthode Fill récupère des données dans une source de données et les utilise pour remplir un DataSet. Mais le comportement par défaut de cette méthode n'inclut aucun transfert des informations de schéma, comme les clés primaires.

Il est intéressant de récupérer les contraintes de clés primaires lorsque les données contenues dans le DataSet doivent être mises à jour à partir de la base de données. Sans ces contraintes, chaque exécution de la méthode Fill ajoutera de nouvelles lignes au DataSet, ce qui peut entraîner la présence de doublons. Dans le cas contraire, si la valeur de la clé primaire d'un enregistrement de base de données est trouvée dans le DataSet, l'enregistrement n'est pas ajouté à ce DataSet : la ligne correspondante est mise à jour avec les

données fraîches.

Ce mécanisme de transfert de la structure d'une table est implémenté dans la méthode `FillSchema` de la classe `SqlDataAdapter`. Celle-ci doit être appelée avant le premier appel à la méthode `Fill`.

```
adapteur.FillSchema(ds, SchemaType.Mapped);
adapteur.Fill(ds, "Eleve");
```

c. Remplissage d'un DataSet sans base de données

Si une grande partie des applications utilisent une base de données pour la persistance des informations qu'elles traitent, d'autres utilisent des informations provenant du Web, de fichiers texte, ou génèrent même leurs propres données. Ces applications aussi peuvent utiliser les classes `DataSet` et `DataTable`. En revanche, elles ne peuvent pas utiliser le type `SqlDataAdapter`, ce qui implique que la création de la structure de données est entièrement à la charge du développeur.

Pour créer une table en mémoire, la première opération à réaliser est la création d'un objet `DataTable`. Son constructeur accepte en paramètre le nom de la table.

```
DataTable table = new DataTable("Eleve");
```

Il faut ensuite créer les différentes colonnes de la table en fournissant des objets de type `DataColumn` à la propriété `Columns` du `DataTable`. Le constructeur de la classe `DataColumn` qui nous intéresse ici accepte deux paramètres : le nom de la colonne et le type de donnée qu'elle doit stocker.

Pour spécifier ce type, il est intéressant d'utiliser l'opérateur `typeof`. Celui-ci prend en paramètre un nom de type .NET et renvoie un objet dont le type est `System.Type`.

```
DataColumn colonneIdentifiant = new DataColumn("Identifiant",
typeof(int));
DataColumn colonneNom = new DataColumn("Nom", typeof(string));
DataColumn colonnePrenom = new DataColumn("Prenom", typeof(string));
DataColumn colonneDateNaissance = new DataColumn("DateNaissance",
typeof(DateTime));

table.Columns.Add(colonneIdentifiant);
table.Columns.Add(colonneNom);
table.Columns.Add(colonnePrenom);
table.Columns.Add(colonneDateNaissance);
```

Il peut être utile de préciser que chaque enregistrement doit avoir une valeur différente pour la colonne *Identifiant*. Pour cela, deux solutions sont à disposition du développeur.

Il est possible de valoriser la propriété `Unique` de l'objet `DataColumn` à `true`. Ainsi, à chaque ajout de donnée, une vérification sera effectuée sur la valeur de l'identifiant fourni. S'il existe déjà, une exception est levée.

Cette solution nécessite que le développeur calcule lui-même la valeur d'identifiant pour chaque enregistrement, ce qui apporte plus de souplesse mais aussi plus de complexité.

Une autre solution est de valoriser la propriété `AutoIncrement` de l'objet `DataColumn` à `true`. Ainsi, à chaque ajout de donnée, c'est l'objet `DataTable` qui est responsable du calcul de l'identifiant. Deux informations peuvent être éditées afin de modifier le calcul : la propriété `AutoIncrementSeed`, qui indique la valeur à utiliser pour le premier enregistrement, et la propriété `AutoIncrementValue` qui définit la valeur de l'incrément à utiliser pour le calcul.

Un identifiant devant, par définition, être toujours renseigné, il est également possible d'utiliser la propriété `AllowDBNull` pour indiquer qu'une valeur de cette colonne peut ou ne doit jamais être égale à `DBNull.Value` (l'équivalent .NET de la valeur `NULL` de SQL).

La création de la variable `colonneIdentifiant` ressemble maintenant à ceci :

```
DataColumn colonneIdentifiant = new DataColumn("Identifiant",
typeof(int));
colonneIdentifiant.Unique = true;
colonneIdentifiant.AutoIncrement = true;
//AutoIncrementSeed vaut par défaut 0 et AutoIncrementValue
//vaut par défaut 1
colonneIdentifiant.AllowDBNull = false;
```

D'un point de vue structurel, il semble logique d'utiliser la colonne *Identifiant* comme clé primaire pour la `DataTable`. La propriété `PrimaryKey` de la classe `DataTable` accepte une valeur de type `DataColumn[]` définissant la liste des colonnes qui composent la clé primaire.

```
table.PrimaryKey = new DataColumn[] { colonneIdentifiant };
```

Certaines propriétés des colonnes concernées sont automatiquement modifiées par cette affectation. Leur propriété `AllowDBNull` est valorisée à `true`, et la propriété `Unique` est également valorisée à `true` dans le cas où un seul champ compose la clé primaire. Dans le cas contraire, la valeur de la propriété n'est pas modifiée.

Une fois cette structure de table créée, il ne reste qu'à insérer des données. Cet ajout est effectué par l'intermédiaire d'objets de type `DataRow`, créés à l'aide de la méthode `NewRow` de la `DataTable`. Cette méthode renvoie une `DataRow` respectant le schéma défini pour la table. Les données de chaque `DataRow` sont stockées dans sa propriété `ItemArray`, de type `object[]`. Il convient d'affecter chacune des données à stocker à l'élément correspondant de ce tableau. Deux possibilités sont offertes pour l'accès à chacun des éléments :

- Accéder à l'index souhaité à travers l'indexeur de la classe `DataRow` :

```
ligne[0] = <valeur>;
```

- Accéder à l'élément à travers l'indexeur de la classe `DataRow` acceptant un nom de colonne :

```
ligne["<Nom de colonne>"] = <valeur>;
```

Chaque enregistrement doit ensuite être ajouté à la collection `Rows` de l'objet `DataTable`.

Le code ci-dessous utilise les deux types d'accès décrits ci-dessus pour l'assignation des données et ajoute les enregistrements créés dans la table.

```
DataRow row1 = table.NewRow();
row1[1] = "MARTIN";
row1[2] = "Henry";
row1[3] = new DateTime(1954, 01, 25);
table.Rows.Add(row1);

DataRow row2 = table.NewRow();
row2[1] = "DUPOND";
row2[2] = "Eric";
row2[3] = new DateTime(1982, 05, 04);
table.Rows.Add(row2);

DataRow row3 = table.NewRow();
row3["Nom"] = "MARTIN";
row3["Prenom"] = "Michel";
row3["DateNaissance"] = new DateTime(2004, 05, 28);
table.Rows.Add(row3);
```

2. Manipulation des données hors connexion

Qu'un `DataSet` ait été rempli à partir d'enregistrements issus de la base de données ou manuellement à partir de données générées par l'application, il est essentiel pour le développeur de pouvoir manipuler son contenu. Pour cela, les classes `DataSet` et associées disposent de nombreuses propriétés et méthodes facilitant la lecture, l'édition ou le tri de données.

a. Lecture des données

La classe `DataTable` fournit différentes manières d'accéder aux données qu'elle contient. La plus simple est d'utiliser sa collection `Rows`. Il est possible de traiter chacun des éléments stockés dans cette collection en utilisant un bloc `foreach`.

```
foreach (DataRow row in table.Rows)
{
    Console.WriteLine("L'identifiant de l'enregistrement est
{0}", row.ItemArray[0]);
}
```

La lecture de chaque élément de l'enregistrement à partir d'un index peut être une opération délicate, notamment si l'ordre des colonnes de la table est modifié. L'utilisation de la méthode d'accès basée sur le nom de la colonne est donc souvent privilégiée.

```
foreach (DataRow row in table.Rows)
{
    Console.WriteLine("L'identifiant de l'enregistrement est
{0}", row["Identifiant"]);
}
```

Un dernier moyen de lire les données d'un objet `DataTable` est de créer un `DataTableReader`, qui est un type dérivé de `DbDataReader` et fournit plusieurs méthodes spécialisées pour la lecture de données avec la

gestion d'un typage fort. La classe `DataTableReader` est utilisée de la même manière que le type `SqlDataReader` (cf. section Sélection de données).

```
DataTableReader dataTableReader = table.CreateDataReader();
while (dataTableReader.Read())
{
    Console.WriteLine("L'identifiant de l'enregistrement est
{0}", dataTableReader[0]);
}
```

b. Création de contraintes

Les contraintes sont une part importante de la gestion de données. Il est en effet souvent nécessaire d'appliquer certaines restrictions sur des jeux de données pour maintenir leur cohérence.

Le framework .NET dispose de deux types de contraintes applicables sur des objets `DataTable` : `UniqueConstraint` et `ForeignKeyConstraint`.

UniqueConstraint

Cette contrainte force l'unicité des valeurs d'un groupe de `DataColumn` : si la contrainte regroupe trois colonnes, la combinaison de ces trois valeurs sera toujours unique. Le non-respect de cette contrainte entraîne le déclenchement d'une exception de type `System.Data.ConstraintException`.

La création de cette contrainte s'effectue en instanciant un objet de type `UniqueConstraint` et en lui passant en paramètre une `DataColumn` ou un tableau d'objets `DataColumn`. Il est ensuite nécessaire de l'ajouter à la collection `Constraints` de la `DataTable`.

```
UniqueConstraint contrainte = new UniqueConstraint();new
DataColumn[] { colonneNom, colonnePrenom }));

table.Constraints.Add(contrainte);
```

Lorsque la contrainte doit être définie sur une seule `DataColumn`, il peut être plus simple de valoriser sa propriété `Unique` à `true` pour obtenir le même résultat.

ForeignKeyConstraint

Les `ForeignKeyConstraint` (ou contraintes de clés étrangères) définissent le comportement des tables liées en cas de modification ou de suppression d'une donnée dans la `DataTable` principale. Les propriétés `UpdateRule` et `DeleteRule` de ce type permettent de définir le type d'action à effectuer à l'aide d'une valeur d'énumération du type `System.Data.Rule` :

- `None` : les lignes associées à la donnée éditée ou supprimée ne sont pas affectées.
- `Cascade` : toutes les lignes connexes sont modifiées ou supprimées.
- `SetDefault` : pour chacune des lignes associées, la donnée est remplacée par la valeur par défaut de la colonne (propriété `DefaultValue` de la `DataColumn`).

- `SetNull` : après application de ce comportement, la valeur des données de la colonne impliquée est `NULL` pour chacune des lignes.

La création d'une contrainte de ce type nécessite l'existence de deux tables. Le premier paramètre du constructeur de la classe `ForeignKeyConstraint` correspond au nom de la colonne dans la table principale, tandis que le second est le nom de la colonne associée dans une table liée. Les données stockées dans ces deux colonnes doivent être du même type.

Le code utilisé pour les exemples précédents a été adapté. Il crée maintenant une `DataTable` supplémentaire nommée *CoursMusique* et ajoute une colonne *FK_Cours* à la table *Eleve* avec les données correspondantes. Une contrainte de clé est ajoutée à cette dernière table afin de référencer correctement l'identifiant d'un cours de musique pour chaque enregistrement de la table. Ici, la table principale est donc *CoursMusique* et la table liée est *Eleve*.

```
//Création de la table CoursMusique
DataTable tableCours = new DataTable("CoursMusique");

DataColumn colonneIdentifiantCours = new
DataColumn("IdentifiantCours", typeof(int));
colonneIdentifiantCours.AutoIncrement = true;
DataColumn colonneIntitule = new DataColumn("Intitule",
typeof(string));
tableCours.Columns.Add(colonneIdentifiantCours);
tableCours.Columns.Add(colonneIntitule);

tableCours.PrimaryKey = new DataColumn[] {
colonneIdentifiantCours };

DataRow rowCours1 = tableCours.NewRow();
rowCours1["Intitule"] = "Piano - Débutant";
tableCours.Rows.Add(rowCours1);

DataRow rowCours2 = tableCours.NewRow();
rowCours2 ["Intitule"] = "Guitare - Débutant";
tableCours.Rows.Add(rowCours2);

DataRow rowCours3 = tableCours.NewRow();
rowCours3 ["Intitule"] = "Solfege";
tableCours.Rows.Add(rowCours3);
```

```
//Création de la table Eleve
DataTable table = new DataTable("Eleve");

DataColumn colonneIdentifiant = new DataColumn("Identifiant",
typeof(int));
colonneIdentifiant.AutoIncrement = true;
DataColumn colonneNom = new DataColumn("Nom", typeof(string));
DataColumn colonnePrenom = new DataColumn("Prenom",
typeof(string));
DataColumn colonneDateNaissance = new DataColumn("DateNaissance",
typeof(DateTime));
DataColumn colonneFKCours = new DataColumn("FK_Cours",
```

```
typeof(int));

table.Columns.Add(colonneIdentifiant);
table.Columns.Add(colonneNom);
table.Columns.Add(colonnePrenom);
table.Columns.Add(colonneDateNaissance);
table.Columns.Add(colonneFKCours);
table.PrimaryKey = new DataColumn[] { colonneIdentifiant };

//Création de la contrainte ForeignKeyConstraint
ForeignKeyConstraint contrainteFK = new
ForeignKeyConstraint(colonneIdentifiantCours, colonneFKCours);
contrainteFK.UpdateRule = Rule.Cascade;
contrainteFK.DeleteRule = Rule.SetNull;
table.Constraints.Add(contrainteFK);

DataRow row1 = table.NewRow();
row1[1] = "MARTIN";
row1[2] = "Henry";
row1[3] = new DateTime(1954, 01, 25);
row1[4] = tableCours.Rows[2]["IdentifiantCours"];
table.Rows.Add(row1);

DataRow row2 = table.NewRow();
row2[1] = "DUPOND";
row2[2] = "Eric";
row2[3] = new DateTime(1982, 05, 04);
row2[4] = tableCours.Rows[2]["IdentifiantCours"];
table.Rows.Add(row2);

DataRow row3 = table.NewRow();
row3["Nom"] = "MARTIN";
row3["Prenom"] = "Michel";
row3["DateNaissance"] = new DateTime(2004, 05, 28);
row3[4] = tableCours.Rows[1]["IdentifiantCours"];
table.Rows.Add(row3);
```

c. Relations entre DataTables

Les relations entre les tables sont un élément important des `DataSet`. Comme les relations de clés étrangères en base de données, elles permettent de naviguer entre les différents objets `DataTable` constituant un stockage de données local. Elles sont très souvent utilisées conjointement avec les contraintes car complémentaires au niveau de l'utilisation.

Créer une relation

Les relations sont représentées par des objets de la classe `DataRelation`. La création d'une relation est effectuée en passant les objets `DataColumn` parent et enfant au constructeur du type. Il est aussi possible de spécifier le nom de la relation afin d'y faire référence plus aisément. Les différentes relations créées pour un `DataSet` doivent être ajoutées à la collection `Relations` de ce conteneur.

Le code suivant ajoute les `DataTable`*Eleve* et *CoursMusique* à un `DataSet`, puis crée une relation nommée *Relation_Cours_Eleve* entre le champ *IdentifiantCours* de la table *CoursMusique* et le champ *FK_Cours* de la table

Eleve.

```
DataSet ds = new DataSet();
ds.Tables.Add(table);
ds.Tables.Add(tableCours);

ds.Relations.Add("Relation_Cours_Eleve", colonneIdentifiantCours,
colonneFKCours);
```

Parcourir les relations

L'utilisation des relations dans un `DataSet` permet d'obtenir tous les enregistrements d'une `DataTable` liés à une `DataRow` d'un autre objet `DataTable`. La méthode `GetChildRows` de la classe `DataRow` effectue cette opération et renvoie les enregistrements dans un tableau de `DataRow`. Elle prend en paramètre le nom de la relation à utiliser pour suivre le lien qu'elle représente.

Le code présenté ci-dessous liste les différents cours de musique existants avec les noms des élèves de chacun.

```
foreach (DataRow ligneCours in ds.Tables["CoursMusique"].Rows)
{
    Console.WriteLine("Les élèves suivants ont choisi le cours
{0}", ligneCours["Intitule"]);

    DataRow[] lignesEleves =
ligneCours.GetChildRows("Relation_Cours_Eleve");

    if (lignesEleves.Length > 0)
    {
        foreach (DataRow ligneEleve in lignesEleves)
        {
            Console.WriteLine("\t{0} {1}", ligneEleve["Prenom"],
ligneEleve["Nom"]);
        }
    }
    else
    {
        Console.WriteLine("------------------ Aucun ------------------");
    }
}
```

Le résultat de son exécution est le suivant :

```
Les élèves suivants ont choisi le cours Piano - Débutant
------------------ Aucun ------------------
Les élèves suivants ont choisi le cours Guitare - Débutant
        Michel MARTIN
Les élèves suivants ont choisi le cours Solfege
        Henry MARTIN
        Eric DUPOND
```

d. État et versions d'une DataRow

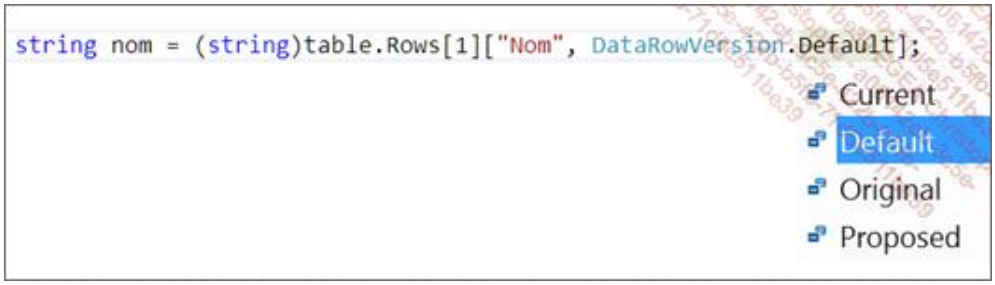
Le type `DataRow` est capable de suivre les modifications apportées à son contenu. Son état est stocké dans sa propriété `RowState`, qui contient une valeur de l'énumération `System.Data.DataRowState`. Les différents états possibles pour une `DataRow` sont au nombre de cinq :

- `Detached` : la ligne n'est pas associée à une `DataTable`.
- `Unchanged` : la ligne n'a pas été modifiée depuis l'appel de la méthode `Fill` ou depuis le dernier appel à sa méthode `AcceptChanges`.
- `Added` : la ligne a été ajoutée à la collection d'enregistrements d'une `DataTable` mais aucun appel à sa méthode `AcceptChanges` n'a encore été effectué.
- `Deleted` : l'objet `DataRow` a été supprimé de la table à laquelle il était associé à l'aide de sa méthode `Delete`.
- `Modified` : au moins une donnée de l'enregistrement a été modifiée et sa méthode `AcceptChanges` n'a pas encore été appelée.

Il est aussi possible d'accéder aux différentes versions d'une ligne à l'aide des quatre valeurs de l'énumération `DataRowVersion`.

- `Original` : version originale de la ligne. Lorsque des modifications sont validées à l'aide de la méthode `AcceptChanges`, la version `Original` est modifiée pour refléter l'état après l'appel de cette méthode. Cette version n'existe pas quand l'état de l'enregistrement est `Added`.
- `Current` : état actuel de la ligne. Lorsque l'enregistrement est à l'état `Deleted`, cette version n'existe plus.
- `Proposed` : lorsque la ligne est en cours de modification, cette version contient les données modifiées mais non validées. À la fin de l'édition, la valeur de cette version est affectée à la version `Current`. Les enregistrements qui ne sont pas attachés à une `DataTable` (état `Detached`) possèdent aussi une version `Proposed`.
- `Default` : l'état par défaut d'une ligne. Lorsque l'état de l'enregistrement est `Added`, `Modified` ou `Deleted`, cette version est équivalente à la version `Current`. Si l'état est `Detached`, elle équivaut à la version `Proposed`.

L'accès à une donnée d'une version spécifique est effectué à partir de l'indexeur implémenté dans la classe `DataRow`. En plus de l'index ou du nom de la colonne, il est possible de spécifier la version de la donnée à manipuler.



La méthode `HasVersion` de la classe `DataRow` permet de savoir si un enregistrement possède un type de version précis. Elle accepte comme paramètre la valeur d'énumération identifiant la version à rechercher et renvoie un booléen. Son utilisation permet d'éviter l'accès à une version de l'enregistrement qui n'existe pas.

```
if (table.Rows[1].HasVersion(DataRowVersion.Original))
{
```

```
    string nomEleve = (string)table.Rows[1]["Nom",
DataRowVersion.Original];
    //Suite du traitement
}
```

e. Modification de données

La modification des données d'un enregistrement est effectuée en affectant des valeurs à ses différents champs. Chaque modification est enregistrée dans la version `Current` de la ligne, et son état est passé à `Modified`. Lorsque plusieurs modifications doivent être apportées, des problèmes de cohérence peuvent apparaître. L'édition d'une colonne faisant partie d'une contrainte d'unicité peut entraîner un doublon temporaire, et une exception est déclenchée alors même que la modification est maîtrisée. Ce type de problème peut être évité en passant la ligne en cours de modification en mode édition.

Cette action est effectuée en appelant la méthode `BeginEdit` de l'objet `DataRow`, ce qui désactive temporairement les vérifications de contraintes sur les valeurs qu'elle contient. Toutes les modifications effectuées dans ce mode sont stockées dans la version `Proposed` de la `DataRow`. Une fois la manipulation des données de l'enregistrement terminée, la validation des modifications est faite avec un appel à `EndEdit`. Il est aussi tout à fait possible d'annuler les changements effectués en utilisant la méthode `CancelEdit`.

Ci-dessous, un exemple de code remplaçant l'identifiant d'une ligne par l'identifiant d'une autre ligne. L'exception n'est pas levée à l'affectation mais à l'appel de la méthode `EndEdit` : la contrainte d'unicité de la clé primaire est désactivée pendant l'édition.

```
DataRow ligneAEditer = table.Rows[1];
ligneAEditer.BeginEdit();
//Sans le mode édition, la ligne suivante déclenche une exception
ligneAEditer["Identifiant"] = table.Rows[2]["Identifiant"];
//mais en mode édition, l'exception est déclenchée
//à l'appel de EndEdit()
ligneAEditer.EndEdit();
```

Le code suivant modifie un nom et valide la modification.

```
DataRow ligneAEditer = table.Rows[2];
Console.WriteLine("Le nom de l'élève est {0}",
ligneAEditer["Nom"]);

ligneAEditer.BeginEdit();
ligneAEditer["Nom"] = "MARTINS";
ligneAEditer.EndEdit();

Console.WriteLine("Le nom de l'élève est maintenant {0}",
ligneAEditer["Nom"]);
```

L'exécution de ce dernier exemple produit le résultat suivant :

```
Le nom de l'élève est MARTIN
Le nom de l'élève est maintenant MARTINS
```

f. Suppression de données

Il existe deux solutions pour qu'une ligne n'existe plus dans une DataTable. Elles ont un comportement légèrement différent l'une de l'autre.

La suppression pure et simple d'un enregistrement est effectuée à l'aide de la méthode Remove de la classe DataRowCollection. Elle prend en paramètre la DataRow à supprimer.

Le code suivant montre comment supprimer la première ligne d'une DataTable en utilisant cette méthode.

```
table.Rows.Remove(table.Rows[0]);
```

La seconde solution est moins radicale : elle marque un enregistrement comme supprimé mais ne l'efface pas tant que les modifications ne sont pas validées au niveau de l'objet DataTable. Si les modifications sont annulées, la ligne reste dans la DataTable et reprend son état précédent. Pour exécuter cette action, il faut utiliser la méthode Delete de l'objet DataRow que l'on souhaite éliminer.

```
table.Rows[0].Delete();
```

L'utilisation de la méthode Delete sur un enregistrement dans l'état Added supprime définitivement l'enregistrement, puisqu'il n'a pas de version d'origine.

g. Valider ou annuler des modifications

La validation de modifications avec la méthode EndEdit des DataRow effectue une validation réversible : il est toujours possible de revenir à la version précédente à l'aide des différentes versions de chacune des lignes.

La méthode AcceptChanges permet de valider définitivement les modifications **locales** d'une DataRow, d'une DataTable ou d'un DataSet. Cette méthode est effectivement implémentée par chacun de ces types, ce qui permet une certaine flexibilité.

Son fonctionnement est le suivant :

- Elle appelle la méthode EndEdit de chacune des lignes en mode édition dans sa portée.
- Pour chaque ligne dans l'état Added ou Modified, elle recopie les valeurs de la version Current vers la version Original. L'état de ces lignes devient alors Unchanged.
- Chaque enregistrement dont l'état est Deleted est supprimé définitivement.

La méthode RejectChanges, quant à elle, annule définitivement chacune des modifications locales effectuées. Comme la méthode AcceptChanges, elle est implémentée au niveau des objets DataRow, DataTable et DataSet. Elle traite les enregistrements d'une manière exactement opposée à la méthode AcceptChanges :

- Elle exécute la méthode CancelEdit de chacune des lignes dans sa portée.
- Elle modifie chacune des lignes à l'état Deleted ou Modified de telle sorte qu'elles reviennent à l'état Unchanged et que les données de leurs version Original et Current soient identiques.
- Les enregistrements à l'état Added sont supprimés définitivement.

Il est à noter que les ForeignKeyConstraint possèdent une propriété AcceptRejectRule permettant de répercuter les modifications à leurs lignes enfants lorsque les méthodes AcceptChanges ou RejectChanges sont appelées.

h. Filtrage et tri à l'aide d'une DataView

Il est possible de modifier la manière dont les informations d'une DataTable sont vues sans pour autant modifier les enregistrements sous-jacents. Pour cela, d'un point de vue orienté base de données, on utilise des vues. Celles-ci ont aussi un équivalent .NET : les DataView. Elles permettent de filtrer ou trier les éléments d'une DataTable sans avoir besoin de recourir à l'envoi de requêtes SQL contenant des clauses WHERE ou ORDER BY vers la base de données. Le traitement permettant de créer ces vues est en effet entièrement local et se base sur un objet DataTable contenant des données. Plusieurs DataView peuvent exister pour le même DataTable, chacune correspondant à un point de vue différent sur les données.

La source de données locale de type DataTable peut être passée en paramètre au constructeur de la classe DataView ou affectée à sa propriété Table. Les propriétés RowFilter et RowStateFilter de la vue filtrent les enregistrements en fonction de prédicats ou de leur état. La propriété Sort permet quant à elle de préciser les champs selon lesquels les enregistrements doivent être ordonnés.

Les enregistrements présents dans une DataView sont présentés sous la forme d'objets de type DataRowView.

RowFilter

Le filtre est défini par une chaîne de caractères indiquant le prédicat qui doit être respecté pour que les lignes soient disponibles dans la DataView. La syntaxe générale de cette chaîne de caractères est similaire à celui de la clause WHERE d'une requête SQL et elle peut être la combinaison de plusieurs conditions à l'aide des mots-clés AND et OR.

Pour obtenir la liste des élèves dont le nom est Martin, on peut utiliser le code ci-dessous.

```
DataView vueFiltree = new DataView(table);
vueFiltree.RowFilter = "Nom = 'MARTIN'";

foreach (DataRowView ligne in vueFiltree)
{
    Console.WriteLine("{0} {1}", ligne["Nom"], ligne["Prenom"]);
}
```

RowStateFilter

Par l'intermédiaire de cette propriété, il est possible de préciser l'état attendu des enregistrements filtrés ainsi que la version qui doit être visible dans la DataView.

Cette propriété est une valeur du type d'énumération DataViewRowState. Les valeurs que ce type expose sont décrites ci-dessous :

- None : aucune ligne.

- `Unchanged` : lignes non modifiées.
- `Added` : lignes ajoutées.
- `Deleted` : lignes marquées comme supprimées.
- `ModifiedCurrent` : lignes modifiées, versions actuelles des données.
- `CurrentRows` : lignes de la version actuelle (non modifiées, nouvelles et modifiées). C'est l'équivalent de la combinaison des valeurs `Unchanged`, `Added` et `ModifiedCurrent`.
- `ModifiedOriginal` : version d'origine des lignes qui ont été modifiées.
- `OriginalRows` : version d'origine des lignes qui n'ont pas été ajoutées. C'est l'équivalent de la combinaison des valeurs `ModifiedOriginal`, `Deleted` et `Unchanged`.

```
DataView vueFiltreeParEtat = new DataView(table);
vueFiltreeParEtat.RowStateFilter = DataViewRowState.Deleted;

Console.WriteLine("Les élèves supprimés sont :");
foreach (DataRowView ligne in vueFiltreeParEtat)
{
    Console.WriteLine("\t{0} {1}", ligne["Nom"],
ligne["Prenom"]);
}
```

L'énumération `DataViewRowState` possède l'attribut `Flags`, ce qui signifie qu'il est possible de combiner plusieurs de ses valeurs pour en former une nouvelle. La valeur qui résulte de cette combinaison permet ainsi d'élargir les critères de recherche et ainsi d'obtenir plus de résultats.

On effectue cette opération en utilisant l'opérateur **OU logique** représenté par le symbole |.

```
vueFiltreeParEtat.RowStateFilter = DataViewRowState.Deleted |
DataViewRowState.Added;
```

La combinaison ci-dessus permet de visualiser dans la `DataView` les lignes dont l'état est `Deleted` ou `Added`.

Sort

Comme pour le filtrage, le tri requiert la valorisation d'une propriété (`Sort`) avec une chaîne de caractères. Cette dernière est similaire à la clause `ORDER BY` d'une requête SQL. Il est ainsi possible de trier les différents élèves par ordre alphabétique de la manière suivante :

```
DataView vueTriee = new DataView(table);
vueTriee.Sort = "Nom ASC, Prenom ASC";

foreach (DataRowView ligne in vueTriee)
{
    Console.WriteLine("{0} {1}", ligne["Nom"], ligne["Prenom"]);
}
```

i. Recherche de données

La recherche dans un jeu de données est facilitée par plusieurs méthodes implémentées dans les classes `DataTable` et `DataView`. Ces méthodes correspondent à deux modes de recherche distinctes : par prédicat ou par clés.

Recherche par prédicat

Ce type de recherche correspond à l'exécution d'une requête `SELECT` dans une base de données. La méthode dans laquelle il est implémenté est d'ailleurs nommée `Select` et est définie dans le type `DataTable`. Elle peut prendre comme paramètres jusqu'à deux chaînes de caractères et une valeur d'énumération `DataViewRowState` représentant respectivement un filtre sur les enregistrements, une clause de tri et l'état des lignes attendues. Ces différents paramètres reprennent la même forme que pour les propriétés `RowFilter`, `RowStateFilter` et `Sort` du type `DataView`.

Cette opération de recherche renvoie un tableau d'objets `DataRow` contenant tous les enregistrements correspondants aux critères définis par l'appel.

```
DataRow[] lesMartin = table.Select("Nom = 'Martin'", "Prenom
ASC");
Console.WriteLine("Les élèves dont le nom est MARTIN sont :");
foreach (DataRow ligne in lesMartin)
{
    Console.WriteLine("\t{0} {1}", ligne["Nom"],
ligne["Prenom"]);
}
```

Recherche par clé

Ce type de recherche permet de récupérer un ou plusieurs éléments dans une `DataView` triée à l'aide des méthodes `Find` et `FindRows`. Il est nécessaire que les enregistrements soient ordonnés car la recherche est effectuée sur la totalité des champs spécifiés dans la propriété `Sort` de la `DataView`. Chacun des critères fournis à l'appel de méthode correspond à la colonne triée positionnée au même index.

Pour éclaircir ce dernier point, considérons que le tri est le suivant : *"Nom ASC, Prenom ASC"*. L'appel aux méthodes de recherche doit fournir une valeur pour le nom et une seconde pour le prénom, dans cet ordre.

La méthode `Find` renvoie une valeur numérique qui est l'index de la première ligne correspondant aux critères de recherche. Si aucun enregistrement n'est trouvé, la valeur de retour est *-1*.

```
DataView vueTrieeParNom = new DataView(table);
vueTrieeParNom.Sort = "Nom ASC";

int index = vueTrieeParNom.Find(new[] { "DURAND" });
if (index == -1)
{
    Console.WriteLine("Aucun élève n'a comme nom DURAND");
}
else
{
    DataRowView ligne = vueTrieeParNom[index];
    Console.WriteLine("Le premier élève donc le nom est DURAND
est {0} {1}", ligne["Prenom"], ligne["Nom"]);
```

```
}
```

La méthode FindRows renvoie quant à elle un tableau d'objets DataRowView.

```
DataView vueTrieeParNomPrenom = new DataView(table);
vueTrieeParNomPrenom.Sort = "Nom ASC, Prenom ASC";

DataRowView[] lignes = vueTrieeParNomPrenom.FindRows(new[] {
"MARTIN", "Michel" });
if (lignes.Length == 0)
{
    Console.WriteLine("Aucun élève ne s'appelle Michel MARTIN");
}
else
{
    Console.WriteLine("Il y a {0} élève(s) dont le nom est Michel
MARTIN", lignes.Length);
}
```

3. Valider les modifications au niveau de la base de données

Jusqu'à présent, toutes les modifications validées ne l'étaient que de manière locale, au niveau des DataRow, DataTable ou DataSet. Mais lorsque les données de ces éléments proviennent d'une base de données, il est possible de transférer les modifications effectuées localement vers la base de données de manière automatique.

Pour cela, il faut faire appel à l'objet SqlDataAdapter qui a été utilisé pour remplir le DataSet avec les enregistrements en provenance de la base de données.

La méthode Update de ce SqlDataAdapter a pour mission de valider les données qui lui sont soumises en exécutant des requêtes SQL d'ajout, de mise à jour ou de suppression définies respectivement par les propriétés InsertCommand, UpdateCommand et DeleteCommand. Ces commandes sont exécutées en fonction de l'état des enregistrements à valider. Si la méthode Update a besoin d'une de ces commandes alors que celle-ci n'a pas été valorisée, une exception est déclenchée.

La validation des modifications apportées sur un jeu d'enregistrements peut être effectuée sur l'ensemble d'un DataSet, sur une DataTable ou sur un nombre restreint de lignes.

Le code suivant crée un DataSet et le remplit à partir d'une base de données. Après les modifications de données, il exécute la méthode Update du SqlDataAdapter afin de valider en base l'ensemble des modifications.

```
var connexion = new
SqlConnection("Server=(LocalDb)\\v11.0;Initial
Catalog=Test;Integrated Security=True");
SqlCommand commande = connexion.CreateCommand();
commande.CommandText = "SELECT * From Eleve;";

SqlDataAdapter adapteur = new SqlDataAdapter();
adapteur.SelectCommand = commande;

DataSet ds = new DataSet();
```

```
adapteur.Fill(ds);

//Placez ici toutes les modifications d'enregistrements

adapteur.Update(ds);
```

En l'état, ce code est parfaitement valide mais génère une exception car les propriétés `InsertCommand`, `UpdateCommand` et `DeleteCommand` ne sont pas valorisées.

a. Générer des commandes de mise à jour automatiquement

La classe `SqlCommandBuilder` est conçue pour générer les requêtes `INSERT`, `UPDATE` et `DELETE` d'un `SqlDataAdapter` à partir de la valorisation de sa propriété `SelectCommand`. Il est donc nécessaire que cette propriété soit valorisée. Pour que le `SqlCommandBuilder` puisse générer des commandes, deux autres restrictions doivent être suivies : les objets `DataTable` concernés par la mise à jour doivent **tous avoir une clé primaire**, et leurs données ne doivent pas être issues d'une jointure entre plusieurs tables de base de données. Si ces conditions ne sont pas respectées, une exception est levée lors de la génération des commandes.

Lors de l'exécution de la méthode `Update`, les valeurs présentes en base de données doivent correspondre à la version `Original` stockée dans le `DataSet`. Si ce n'est pas le cas, une exception de type `DBConcurrencyException` est levée.

Nous verrons comment éviter ce type d'exceptions à la section Gestion des accès concurrentiels de ce même chapitre.

Pour utiliser le type `SqlCommandBuilder`, il suffit de l'instancier en passant comme paramètre un `SqlDataAdapter` à partir duquel les commandes seront générées. Pour visualiser la requête encapsulée dans une de ces commandes, il faut utiliser les méthodes `GetInsertCommand`, `GetUpdateCommand` ou `GetDeleteCommand`. Celles-ci renvoient toutes un objet `DbCommand` dont la propriété `CommandText` représente la requête SQL associée.

Cet exemple reprend le code précédent et ajoute les instructions nécessaires à la génération et à l'affichage de commandes pour la variable `adapteur`.

```
var connexion = new
SqlConnection("Server=(LocalDb)\\v11.0;Initial
Catalog=Test;Integrated Security=True");
SqlCommand commande = connexion.CreateCommand();
commande.CommandText = "SELECT * From Eleve;";

SqlDataAdapter adapteur = new SqlDataAdapter();
adapteur.SelectCommand = commande;

DataSet ds = new DataSet();
adapteur.Fill(ds);

//Placez ici toutes les modifications d'enregistrements

//Création du SqlCommandBuilder et association au SqlDataAdapter
SqlCommandBuilder commandBuilder = new
SqlCommandBuilder(adapteur);
```

```
string requeteInsert =
commandBuilder.GetInsertCommand().CommandText;
string requeteUpdate =
commandBuilder.GetUpdateCommand().CommandText;
string requeteDelete =
commandBuilder.GetDeleteCommand().CommandText;

Console.WriteLine("Requête INSERT :");
Console.WriteLine(requeteInsert);
Console.WriteLine();
Console.WriteLine("Requête UPDATE :");
Console.WriteLine(requeteUpdate);
Console.WriteLine();
Console.WriteLine("Requête DELETE :");
Console.WriteLine(requeteDelete);

adapteur.Update(ds);
```

Cet exemple produit l'affichage suivant :

```
Requête INSERT :
INSERT INTO [Eleve] ([Nom], [Prenom], [DateNaissance]) VALUES
(@p1, @p2, @p3)

Requête UPDATE :
UPDATE [Eleve] SET [Nom] = @p1, [Prenom] = @p2, [DateNaissance] =
@p3 WHERE (([Identifiant] = @p4) AND ([Nom] = @p5) AND ([Prenom] =
@p6) AND ((@p7 = 1 AND [DateNaissance] IS NULL) OR ([DateNaissance] =
@p8)))

Requête DELETE :
DELETE FROM [Eleve] WHERE (([Identifiant] = @p1) AND ([Nom] =
@p2) AND ([Prenom] = @p3) AND ((@p4 = 1 AND [DateNaissance] IS
NULL) OR ([DateNaissance] = @p5)))
```

L'objet `SqlCommandBuilder` ne génère une commande que si sa valeur est `null`. En d'autres termes, si une commande d'insertion, de mise à jour ou de suppression est valorisée manuellement, elle ne sera pas écrasée par une commande générée.

b. Commandes de mise à jour personnalisées

La génération de commande ne convient pas à tous les cas d'utilisation. Il est en effet très courant, notamment dans les applications d'entreprises, de marquer certains enregistrements comme supprimés au lieu de les supprimer définitivement ou d'enregistrer certaines informations dans une table d'historisation à chaque modification de base de données. Une solution pour cela est de définir des commandes personnalisées permettant ainsi d'exécuter des requêtes SQL adaptés aux besoins.

Pour effectuer cette action, il suffit de créer une commande et de l'affecter à la propriété correspondante de l'objet `SqlDataAdapter`.

c. Gestion des accès concurrentiels

Les applications sont de plus en plus souvent multi-utilisateurs, ce qui implique de plus en plus d'accès concurrentiels. Deux optiques peuvent être utilisées pour gérer les conflits qui en découlent : le verrouillage pessimiste et le verrouillage optimiste.

Le verrouillage pessimiste est le plus contraignant pour les utilisateurs. Son mode de fonctionnement est simple : lorsqu'un enregistrement est manipulé par un utilisateur, il est verrouillé et ne peut alors pas être modifié par un autre utilisateur. Ce verrou n'est débloqué que lorsque l'utilisateur travaillant sur les données a validé ses modifications. Ce mode de fonctionnement a donc pour objectif d'empêcher tout conflit.

Quant au verrouillage optimiste, ce n'est en fait pas vraiment un verrouillage. Il correspond à une approche visant à la résolution des conflits plutôt qu'à leur évitement. Les utilisateurs sont donc libres d'éditer les données simultanément. À la validation effective des mises à jour, une comparaison est effectuée avec les données correspondantes en base de données. Si la version `Original` des données locales est différente des données en base, alors un autre utilisateur a pu modifier ces données et on est donc dans un cas de conflit.

Cette approche est utilisée dans les commandes de mise à jour et de suppression générées automatiquement par le `SqlCommandBuilder`.

Trois solutions sont disponibles pour gérer les mises à jour conflictuelles :

- Abandonner la mise à jour.
- Écraser les données en base avec les données plus récentes.
- Demander l'avis de l'utilisateur.

La première solution est celle qui est utilisée par l'objet `SqlDataAdapter`. L'exécution de la requête de mise à jour renvoie un entier indiquant le nombre d'enregistrements modifiés. Si cette valeur est à 0 lorsqu'une ligne modifiée est en cours de validation, l'objet `SqlDataAdapter` sait qu'aucune ligne correspondant aux critères définis par la requête n'a été trouvée. Par défaut, une exception est levée dans ce cas. Ce comportement peut être modifié en affectant la valeur `true` à la propriété `ContinueUpdateOnError` du `SqlDataAdapter`.

Il est possible de contrôler plus finement le comportement de l'application en cas de conflit en utilisant l'événement `RowUpdated` de la classe `SqlDataAdapter`. Cet événement, déclenché juste après chaque exécution d'une requête de mise à jour, fournit les éléments nécessaires à l'analyse du résultat de la mise à jour et au contrôle de l'exécution des mises à jour suivantes.

Le délégué définissant la signature du gestionnaire de l'événement `RowUpdated` est le suivant :

```
public delegate void SqlRowUpdatedEventHandler(object sender,
SqlRowUpdatedEventArgs e);
```

L'objet de type `SqlRowUpdatedEventArgs` passé en paramètre au gestionnaire d'événements possède une propriété `Row` permettant de connaître la ligne qui vient d'être traitée ainsi que les deux propriétés `Command` et `StatementType` contenant respectivement la commande SQL exécutée et une valeur d'énumération représentant le type de requête envoyée vers la base de données.

D'autres propriétés de cet objet fournissent une représentation du résultat de l'exécution de la commande. C'est le cas des propriétés `RowCount` et `RecordsAffected` qui contiennent respectivement le nombre de lignes devant être mises à jour et le nombre d'enregistrements réellement affectés. La propriété `Errors` contient l'exception potentiellement déclenchée lors de l'exécution de la commande.

Enfin, la propriété `UpdateStatus` permet de connaître le comportement qui va être adopté par l'application

après le traitement de l'enregistrement en cours, mais elle donne aussi la possibilité au développeur de modifier ce chemin d'exécution, notamment en cas d'erreur. Les valeurs disponibles pour cette propriété sont les suivantes :

- `Continue` : l'exécution continue en l'état.
- `ErrorsOccured` : des erreurs se sont produites, il est nécessaire de les traiter.
- `SkipCurrentRow` : la mise à jour de la ligne en cours est abandonnée, mais l'exécution peut être poursuivie pour les autres enregistrements.
- `SkipAllRemainingRows` : la ligne en cours de traitement et tous les enregistrements qui la suivent ne doivent pas être mis à jour.

Le code ci-dessous définit un gestionnaire d'événements affichant l'identifiant des enregistrements dont la mise à jour n'a pas pu être exécutée.

```
void adapteur_RowUpdated(object sender, SqlRowUpdatedEventArgs
args)
{
    if (args.RecordsAffected == 0)
    {
        Console.WriteLine("La ligne dont l'identifiant est {0}
n'a pas été mise à jour.", args.Row["Identifiant"]);
        args.Status = UpdateStatus.SkipCurrentRow;
    }
}
```