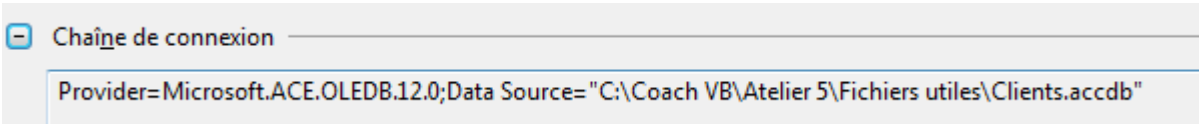
La chaîne de connexion

Vous retrouvez le format de la chaîne de connexion vu lors de la configuration de la source de données avec l'Assistant basé sur des couples **Clé = Valeur** :

```
connectionString="Provider=Microsoft.ACE.OLEDB.12.0;  Data source
=|DataDirectory| \Clients.accdb"
```

Avez-vous remarqué que cette chaîne n'est pas tout à fait identique à celle que vous aviez obtenue au moment de la configuration avec l'Assistant à savoir :



L'attribut Data Source n'est plus tout à fait le même car vous avez décidé dans une étape ultérieure de copier le fichier de données dans le dossier du projet. Du coup, Visual Studio a mis automatiquement à jour la chaîne de connexion pour qu'elle s'appuie non plus sur le chemin initial mais sur le dossier de projet.

Que signifie |DataDirectory| ?

Il s'agit d'une chaîne de substitution pour indiquer dynamiquement le chemin du fichier de données. Par défaut il représente le répertoire de l'application.

A noter que si la chaîne de connexion doit comporter des informations sensibles telles que des paramètres de sécurité, il est possible de la chiffrer pour une meilleure protection de l'information.

La chaîne de connexion, tout comme n'importe quel paramètre que vous souhaitez configurer dans le fichier de configuration de l'application, est également éditable de façon plus conviviale qu'au format XML via le Concepteur de projet de l'application.

La chaîne de connexion est bien là ! Elle est directement modifiable de façon statique depuis cet écran qui est le reflet exact des paramètres personnels que vous pouvez ajouter au fichier de configuration.

Nous verrons dans la suite de cet atelier comment utiliser cette même information par code.

Le Concepteur de DataSet vous donne une représentation visuelle et des outils pour manipuler les objets qui constituent le DataSet que vous avez généré avec l'Assistant.

DataSet

En réalité, le groupe de données que nous avons généré correspond évidemment à du code généré et ajouté au projet. Pour le voir, il suffit de cliquer l'icône Afficher tous les fichiers  dans la barre d'outils de l'Explorateur de solutions.

Le fichier qui contient le code généré par le Concepteur (Designer en anglais) de DataSet est le fichier d'extension *.Designer.vb.

Notez que le contenu de ce fichier est écrasé chaque fois qu'une modification est effectuée depuis la surface de design du Concepteur.

Les deux autres fichiers ont pour extension :

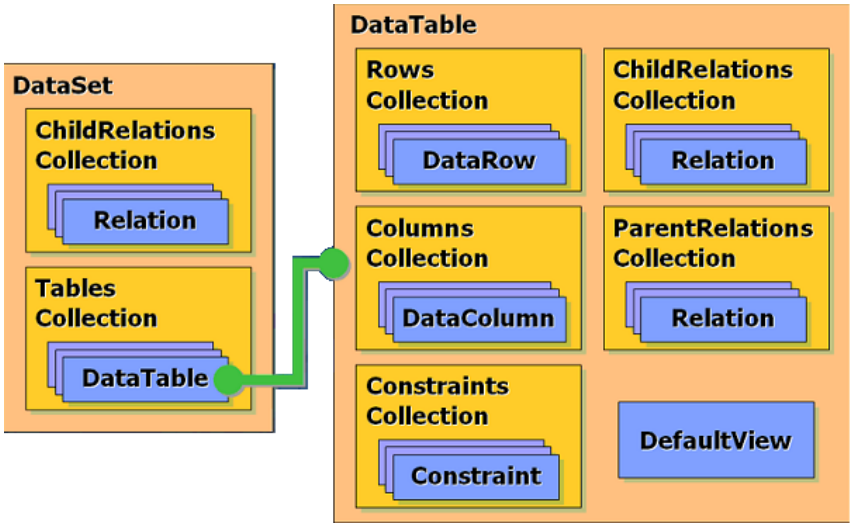
- *.xsc : ce fichier stocke vos préférences utilisateurs au niveau de la configuration de la source de données.
- *.xss : ce fichier stocke les informations du Concepteur relatives aux objets du DataSet, telles que leur emplacement ou leur taille.

Si vous voulez définir votre propre code et personnaliser le DataSet, il peut y avoir un troisième fichier d'extension *.vb qui contient vos définitions dans une classe partielle. (Pour le générer, il suffit de faire un clic droit sur la surface du Concepteur de DataSet > Afficher le code).

Le concepteur affiche deux objets : un objet Contacts et un objet ContactsTableAdapter.

A quoi correspondent ces objets ?

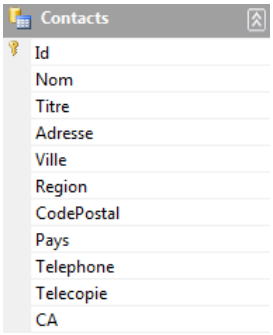
Penchons-nous (pas trop quand même pour ne pas tomber) sur la structure d'un groupe de données (DataSet). Elle est très proche de celle d'une base de données relationnelle dans la mesure où un Dataset est constitué d'une hiérarchie d'objets représentant des tables (DataTable), des lignes (DataRow), des colonnes (DataColumn), des contraintes (Constraint) et même des relations (Relation) :



La seule chose importante à comprendre est qu'on travaille ici en mémoire, et qu'on manipule des objets. C'est pourquoi il n'y aucun lien entre un DataSet et ses sources de données physiques.

L'objet Contacts représente l'une des tables que peut contenir votre DataSet. Dans notre cas, nous n'avons qu'une seule table car nous n'avons sélectionné qu'un seul objet de la base au moment de la configuration de la source de données.

Avez-vous remarqué que l'objet DataTable s'appelle Contacts comme la table dans la base de données ?



Non seulement il s'appelle Contacts, mais en plus il contient une collection de colonnes (d'objets DataColumn) dont les noms sont le reflet des noms de colonnes dans la base de données. Pratique non ?

En fait, notre DataSet a la particularité d'être typé.

C'est quoi un DataSet typé ?

Comme son nom l'indique, cela veut dire qu'il est typé…ah ouais ! On est bien avancé avec ça…

Plus sérieusement, il est vraiment question de type de données. L'idée est de vous aider à manipuler les données le plus simplement possible.

En théorie, un DataSet contient une collection de DataTable qui contiennent elles-mêmes des collections de lignes (DataRow) et colonnes (DataColumn). Cela veut dire que pour accéder à un élément de données, il vous faut manipuler des écritures du type tabulaire :

Tables(indexe).Rows(indexe de la ligne)(indexe de la colonne)

et en plus vous récupéreriez une donnée dont vous ignorez le type exact car le modèle vous retourne une donnée de type Object.

L'intérêt du DataSet typé que nous a généré l'Assistant est qu'il est plus riche qu'un DataSet normal dans la mesure où il contient des objets nommés et dont le type reflète rigoureusement les types des données en base.

Exemple :

- pour manipuler la table de notre DataSet, au lieu d'utiliser un objet retourné par l'écriture Tables(0), on va pouvoir directement travailler avec l'objet DataTable nommé Contacts.
- Pour accéder à une information dans la colonne Nom, on va bénéficier d'une propriété d'objet appelée Nom dont le type est String.

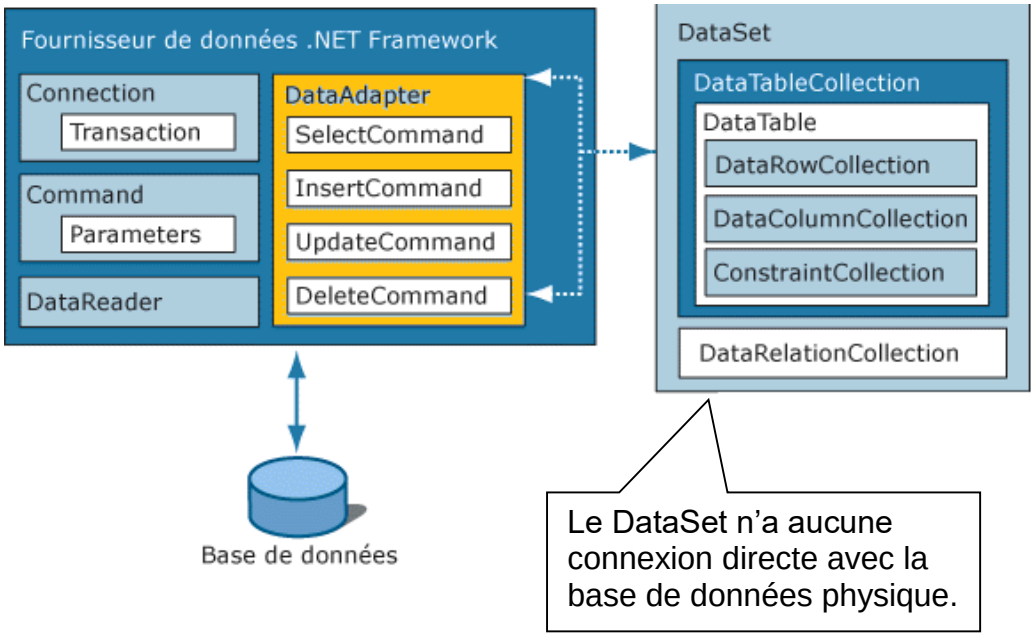
le code ne va s'en trouver que plus lisible et en plus on va bénéficier de l'aide précieuse de l'IntelliSense et des indications du compilateur en cas d'erreur de manipulation au moment de la compilation !

En résumé, le premier objet Contacts va donc nous servir à recueillir en mémoire des données de la table Contacts en provenance de la source de données.

Et le deuxième objet ContactsTableAdapter ?

L'objet TableAdapter est celui qui permet de relier la source de données physique à l'objet mémoire DataSet. Et oui, car le DataSet est indépendant de la source de données. Il ne la connait même pas ! C'est cet autre objet qui fait tout le travail de communication avec la base, à la fois de connexion et à la fois de requêtage pour extraire ou mettre à jour les données.

C'est un objet qui comprend 4 sous-objets de commande, chargés de « commander » la base, chacun dédié à un type de requête vers la base. L'objet SelectCommand gère l'extraction de données tandis que les trois autres objets s'occupent des autres types de requêtes (Insert, Update et Delete).



Dans le schéma ci-dessus, on parle de DataAdapter et non de TableAdapter. Qu'est-ce que c'est que cette embrouille encore ?

Dit simplement, retenez qu'un TableAdapter est un objet DataAdapter mais plus riche que l'objet de base. Il est généré pour nous par l'Assistant Configuration de sources de données pour nous permettre d'aller plus loin en matière d'interaction avec la base de données.

Ce qui va nous intéresser plus encore, ce sont les méthodes de l'objet qui vont nous permettre de travailler sur les données dans la suite de l'atelier, telles que Fill, Update, Insert et Delete :

```
1035
1036    Public Property ClearBeforeFill ...
1045
1046    Private Sub InitAdapter ...
1158
1159    Private Sub InitConnection ...
1164
1165    Private Sub InitCommandCollection ...
1174
1175    Public Overloads Overridable Function Fill ...
1186
1187    Public Overloads Overridable Function GetData ...
1196
1197    Public Overloads Overridable Function Update ...
```

La méthode Fill par exemple, est celle qu'il va nous falloir utiliser pour extraire les données de la base de données et les charger dans l'objet DataSet.

Comment la méthode Fill fait-elle pour interroger la base de données ?

Elle s'appuie sur l'objet SelectCommand contenu dans le TableAdapter qui définit la requête SELECT à exécuter sur le moteur de base de données.

Pour voir les quatre objets de commande et le détail des requêtes générées de l'objet ContactsTableAdapter, sélectionnez l'objet dans le Concepteur de DataSet puis tapez la touche F4 pour faire apparaître la fenêtre de propriétés :

De quoi avons-nous besoin ? Sur la base de ce que nous avons vu à l'exercice précédent, il nous faut :

- L'objet ContactsTableAdapter pour envoyer l'ordre d'extraction des données à la base de données.
- L'objet Contacts pour stocker les données en mémoire en retour de l'extraction.

Ensuite, il ne nous restera qu'à charger les contrôles d'affichage en s'appuyant sur le mécanisme de liaison de données (databinding) que nous avons vu à l'atelier précédent. Et oui, on retombe exactement sur le même fonctionnement puisque l'objet Contacts dans lequel nous allons charger les données est aussi un objet de type DataTable au même titre que l'objet newDataTable que nous avions utilisé pour lire les données du fichier texte.

N'oubliez pas qu'un DataSet en lui-même ne contient pas de données directement. Les données sont en fait chargées dans la collection de DataTable qu'il contient.

Dans notre cas, le DataSet ne contient qu'une seule table (Contacts) et comme il est typé nous pouvons accéder directement à l'objet DataTable via l'écriture ClientsDataSet.Contacts. Pratique non ?

Peut-être vous attendiez-vous à une écriture du type :

DataTable en retour = TableAdapter.Fill() ?

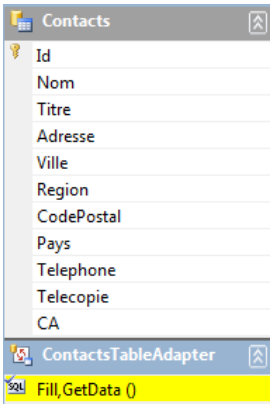
Pourquoi pas mais il se trouve que la méthode Fill attend le conteneur de données en paramètre de la procédure. Le paramètre est passé par référence c'est-à-dire que les modifications effectuées dans le traitement de la procédure sont directement appliquées à l'objet initial passé en paramètre.

Que fait la méthode Fill concrètement ?

La méthode Fill récupère des lignes de la source de données à l'aide de l'instruction SELECT définie dans l'objet SelectCommand.

Pourquoi est-ce qu'il ne faut pas ouvrir une connexion sur la base de données avant d'appeler la méthode Fill ?

Tout simplement, vous vous en doutez, parce que c'est la méthode qui s'occupe de tout ☺. Si une connexion est déjà ouverte avant l'appel à la méthode Fill, elle est utilisée. Sinon, elle est automatiquement ouverte le temps de récupérer les données. La connexion est aussitôt fermée une fois les données rapatriées dans le DataSet. Si la connexion était ouverte au préalable, elle reste ouverte.



Par défaut un TableAdapter comprend deux méthodes pour extraire les données d'une base de données afin de les insérer dans un DataTable :

- La méthode Fill, qui utilise un DataTable existant comme paramètre et le remplit.
- La méthode GetData qui renvoie un nouveau DataTable déjà rempli.

Dans notre cas, c'est bien la méthode Fill la plus appropriée puisque notre DataSet typé contient un objet DataTable tout prêt pour réceptionner les données.

Que va-t-il falloir faire pour enregistrer les données ?

Est-ce que vous vous souvenez comment fonctionne la grille de données ?

Nous avons vu que la grille de données est capable d'enregistrer les modifications effectuées par l'utilisateur automatiquement dans sa source de données (DataTable). C'est le principe de la liaison de données (ou databinding) que l'on dit bidirectionnel puisque cela marche aussi bien dans un sens que dans l'autre.

Donc, c'est dans l'objet DataTable qui constitue la source de données de la grille qu'il faut récupérer les changements pour les appliquer à la base de données.

Et pour discuter avec la base il nous faut …?

L'objet de type TableAdapter qui propose une méthode nommée Update pour répercuter les changements dans la base de données.

Alors allons-y !

Que fait la méthode Update concrètement ?

Elle analyse chaque modification apportée dans la source de données passée en paramètre et exécute la commande appropriée pour propager la ligne modifiée à la base de données :

- s'il s'agit d'une nouvelle ligne, elle exécute la commande INSERT,
- s'il s'agit d'une suppression de ligne, elle exécute la commande DELETE,
- et s'il s'agit d'une ligne ayant subi une modification, elle exécute la commande UDPATE.

Gérer les erreurs

La mise à jour de données est plus délicate que l'extraction dans le sens où des erreurs peuvent se produire, voire des conflits.

Pour éviter que des messages intempestifs apparaissent à l'utilisateur, il faut gérer les erreurs potentielles dans le code. Pour cela, on utiliser une structure de décision Try…Catch…End Try :

- Le principe consiste à précéder le bloc d'instructions dans lequel une erreur peut se produire de l'instruction Try.
- Ensuite on peut ajouter autant de bloc Catch que de type d'exception générée par le système, à condition que vous sachiez les anticiper. Si vous ne savez pas à l'avance le type d'erreur possible, alors un seul bloc Catch fera l'affaire.

C'est quoi une Exception ?

Lorsqu'une erreur se produit, le Framework .NET génère une sorte d'alerte pour prévenir l'application qu'un problème est survenu pendant l'exécution. Cette erreur peut être analysée dans le code à l'aide d'un objet qui détaille précisément le problème (source de l'erreur, message de l'erreur etc…) qu'on appelle Exception. D'autres classes dérivées de la classe de base caractérisent des types d'exception particuliers (System.OverflowException, System.IO.FileNotFoundException etc…)

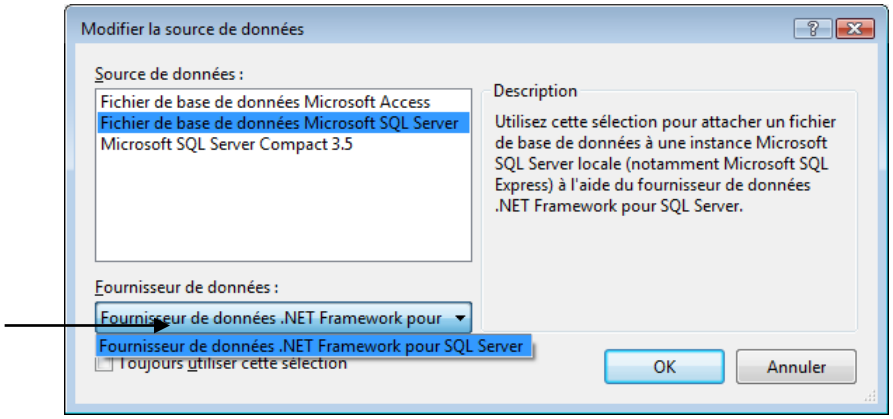
Du coup, pour valider ou annuler tous les changements d'un groupe de données, le Framework.NET nous fournit deux méthodes, respectivement AcceptChanges et RejectChanges. Ces méthodes peuvent s'appliquer sur le

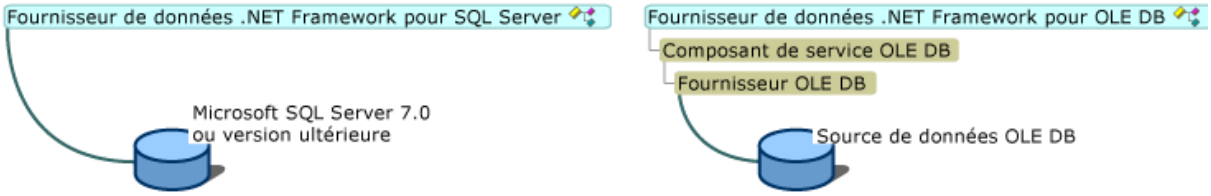
DataSet entier (s'il contient plusieurs tables) ou sur une table DataTable particulière, ou même sur une ligne DataRow.

Notez que la méthode AcceptChanges est appelée automatiquement à la fin du processus de traitement de la méthode Update du TableAdapter pour marquer toutes les lignes à l'état Unchanged.

Fournisseur de données

Il faut savoir que le fournisseur de données pour SQL Server fourni dans le Framework.NET est très léger et donc très performant. Il est en effet optimisé de façon à accéder directement à SQL Server, sans ajout d'une couche OLEDB ou ODBC comme c'est le cas pour le fournisseur que nous avons utilisé à l'exercice précédent pour accéder à la base Access :





Que contient cette chaîne de connexion ?

Elle est composée de plus de couples Clé/Valeur que celle pour Access. Chaque fournisseur a en effet ses spécificités.

Ici, on retrouve :

- Data Source : cette clé détermine le nom de l'instance SQL Server dans laquelle est prise en charge la base de données. Une instance de SQL Server Express Edition est une instance nommée donc elle suit le format :

<Nom de la machine>\<Nom de l'instance>

Le nom de la machine peut être substitué par un point et le nom de l'instance est SQLEXPRESS.

- AttachDbFilename : cette clé indique le nom du fichier de base de données d'extension *.mdf précédé du chemin d'accès complet. Grâce à cette clé, vous n'avez pas à vous soucier d'attacher le fichier de données au moteur SQL Server via la console de gestion de SQL Server. Le Framework .NET va attacher lui-même la base au serveur de manière automatique à chaque exécution de l'application. Pratique non ?
- Integrated Security : cette clé indique que vous vous appuyez sur l'authentification Windows. Mais alors comment est-ce qu'un compte utilisateur local va être à même d'attacher dynamiquement la base de données au moteur SQL Server Express ?

En fait, c'est grâce à une nouvelle fonctionnalité de SQL Server Express qu'on appelle instances utilisateur. Il s'agit d'une instance séparée du moteur de base de données qui est générée automatiquement et qui permet donc à l'utilisateur de se connecter sans avoir de privilèges administrateurs système.

- User Instance : cette clé indique que vous allez justement utiliser une instance utilisateur de SQL Server Express. Elle va être générée par l'instance parent que vous avez indiqué dans la clé Data Source.
- Connect Timeout : indique le temps maximum (en secondes) pour établir la connexion. Au-delà, la connexion échoue.