

ADO.NET

Ado.net est une technologie d'accès aux données qui fournit un ensemble des classes permettant d'accéder aux données relationnelles. Même si Microsoft a repris le nom d'ADO, déjà présent en VB6, la stratégie d'accès aux données est totalement différente; on retrouve néanmoins une certaine facilité d'accès.

Ado.net propose deux modes d'accès, le **mode connecté** et le **mode déconnecté**.

Le mode connecté.

Ce mode classique maintient la connexion à la base de données, il permet de créer un mécanisme de "curseur" permettant de parcourir les données ligne à ligne. Ado.net ne propose qu'un accès en lecture -en avant seulement- avec ce mode. Il est approprié pour parcourir des tables volumineuses rapidement.

Le mode déconnecté.

C'est la grande nouveauté de l'architecture .net. Après une connexion et le chargement de données, tout se passe en mémoire. Ceci libère les ressources du serveur de données, par contre le mécanisme de cohérence et d'intégrité des données est exigeant.

Nous allons étudier ce dernier type d'accès.

ADO en mode déconnecté.

Dotnet propose un ensemble de classes. Une classe générique, **DataSet**, est au centre du dispositif ; c'est cette classe qui servira de *conteneur en mémoire* des tables et des requêtes. Par ailleurs, en plus du framework ADO, l'environnement Visual Studio propose des composants graphiques qui facilite l'accès aux données, même si certains mécanismes sont ainsi masqués.

Nous allons partir d'un cas pour décrire les classes mises en oeuvre.

Présentation du cas.

Nous allons mettre en oeuvre ces notions à partir d'un exemple très simplifié de gestion d'une auto-école :

ConduiteAuto93 est une auto-école récemment installée en Seine-St-Denis, elle propose des forfaits incluant un nombre de leçons dépendant du forfait choisi, des séances de révision du code et une inscription au permis. ConduiteAuto dispose de six véhicules. Chaque leçon dure une ou deux heures.

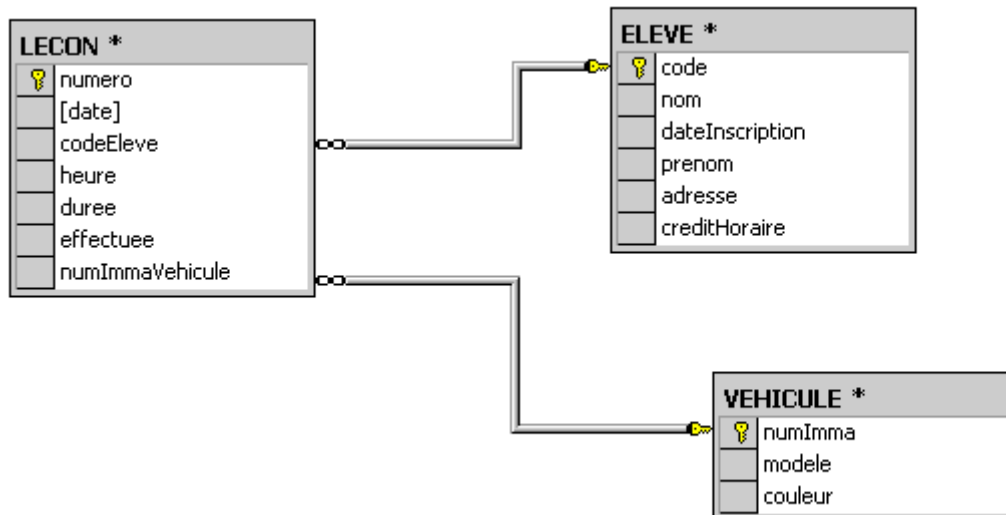


fig. 1 Schéma relationnel

Remarques :

- Le champ *effectuee* dans la table *Leçon* prendra la valeur "vrai" si la leçon a réellement eu lieu
- Le champ *creditHoraire* de la table *Elève* enregistre les heures de conduite restantes, prévues dans le forfait. A chaque leçon suivie ce champ est mis à jour.

La base de données est sous SQL-Server.

Différents traitements seront proposés dans différents formulaires. Le formulaire d'accueil se présente ainsi :

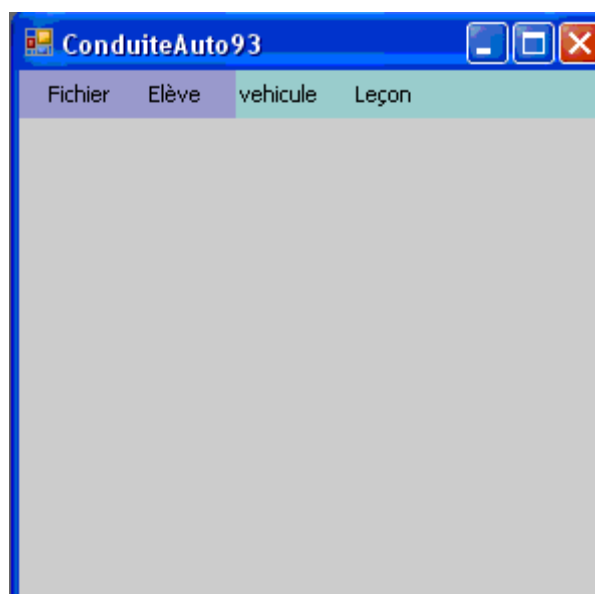


fig. 2 formulaire d'accueil

Télécharger les fichiers (base SQL à restaurer et application Visual Studio vide)

● La connexion à la base de données

La première étape est de créer une connexion à la base de données : la base de données *EcoleAuto93* créée sous SQL-Server.

Visual Studio propose de créer une source données à l'aide d'un assistant : Données/Ajouter une source de données.



fig. 3 : ajout d'une source de données

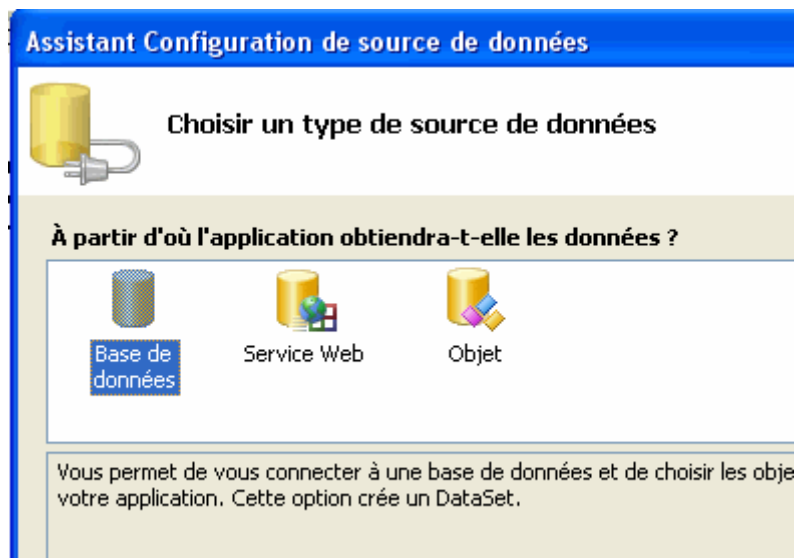


fig. 4 : assistant d'ajout d'une source de données

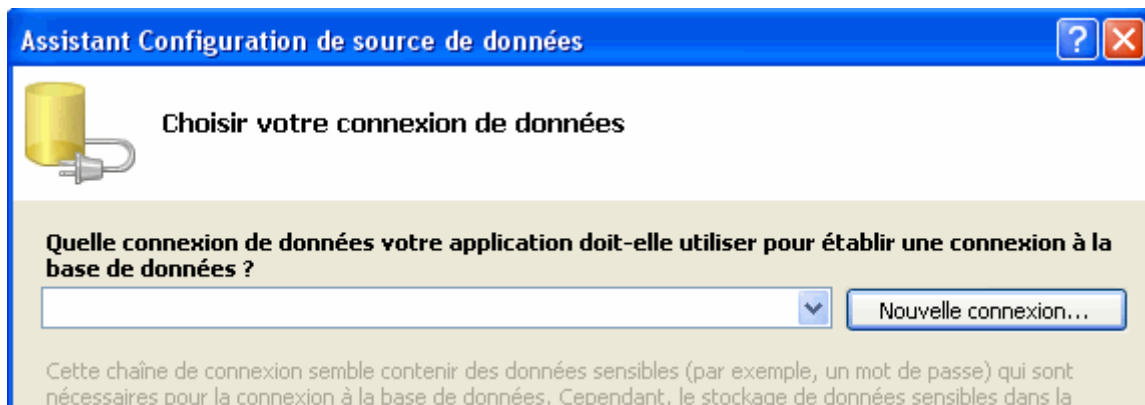


fig. 5 : assistant d'ajout d'une source de données

Définir la nouvelle source de données :

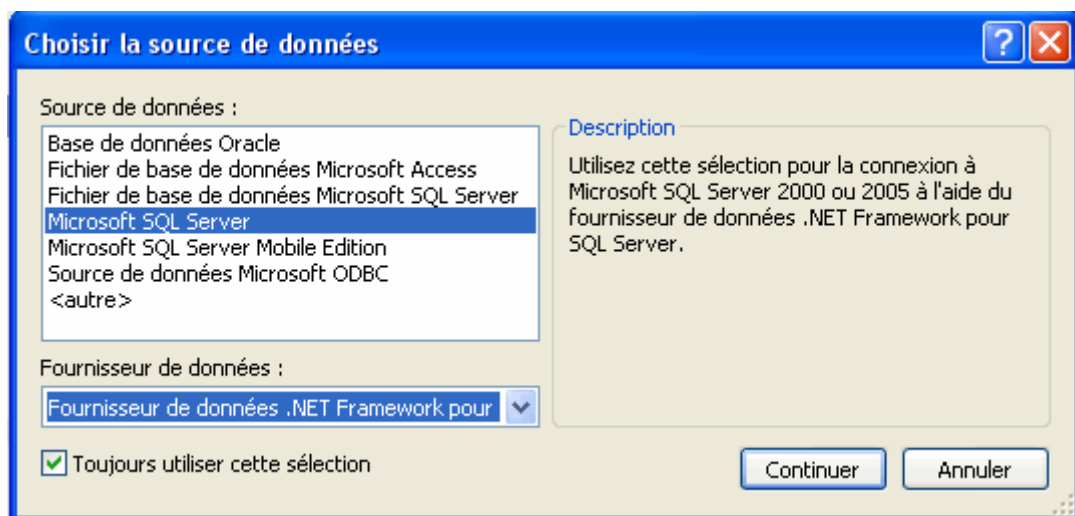


fig. 6 : définir la base de données

Configurer la source de données :

Ajouter une connexion

Entrez les informations pour vous connecter à la source de données sélectionnée ou cliquez sur "Modifier" pour sélectionner une autre source de données et/ou un autre fournisseur.

Source de données :
Microsoft SQL Server (SqlClient) [Modifier...]

Nom du serveur :
PATRICE [Actualiser]

Connexion au serveur

☒ Utiliser l'authentification Windows
☐ Utiliser l'authentification SQL Server

Nom d'utilisateur : []
Mot de passe : []
☐ Enregistrer mon mot de passe

Connexion à la base de données

☒ Sélectionner ou entrer un nom de base de données :
autoEcole []
☐ Attacher un fichier de base de données :
[] [Parcourir...]
Nom logique : []

[Avancées...]

[Tester la connexion] [OK] [Annuler]

fig. 7 : configuration de la source données

Tester la connexion et faire ok

Ceci termine la configuration de cette nouvelle source de données. Il faudra juste indiquer ensuite que la connexion pour cette application utilise cette source données :

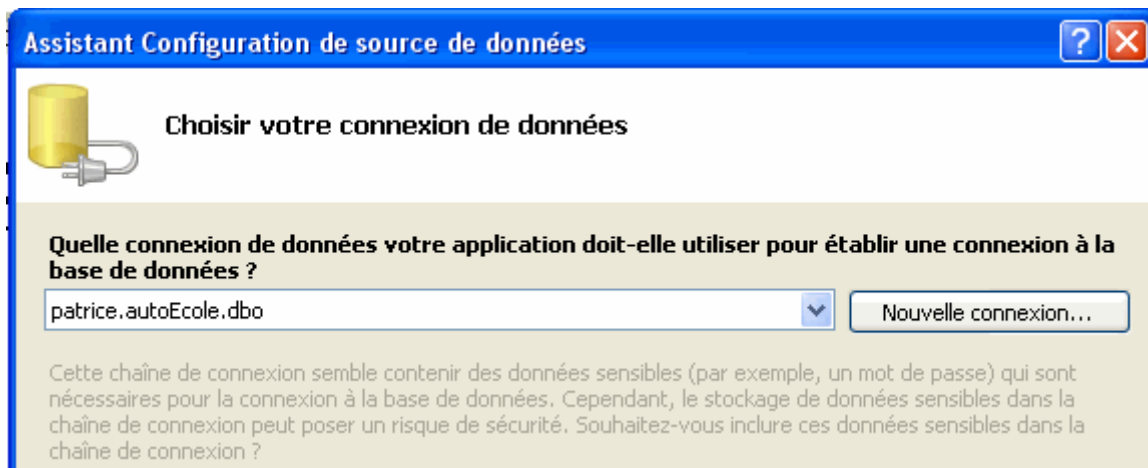


fig. 8 : association entre la connexion et la source de données

Valider la demande de création de la chaîne de connexion :

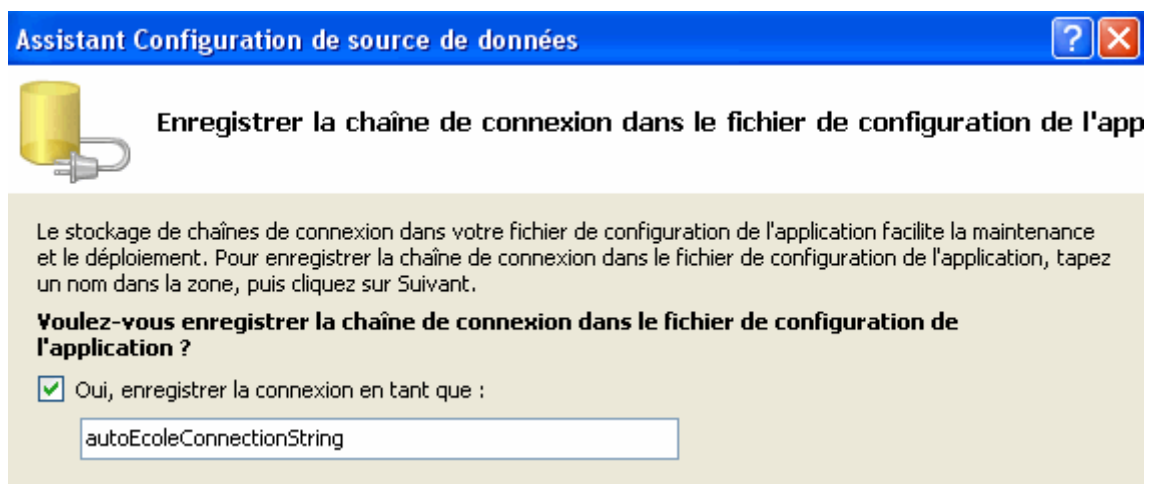


fig. 9 : enregistrement de la chaîne de connexion

Terminer en indiquant les objets de la bases importés -la base contient aussi une procédure stockée- dans le DataSet, ainsi que le nom choisi.

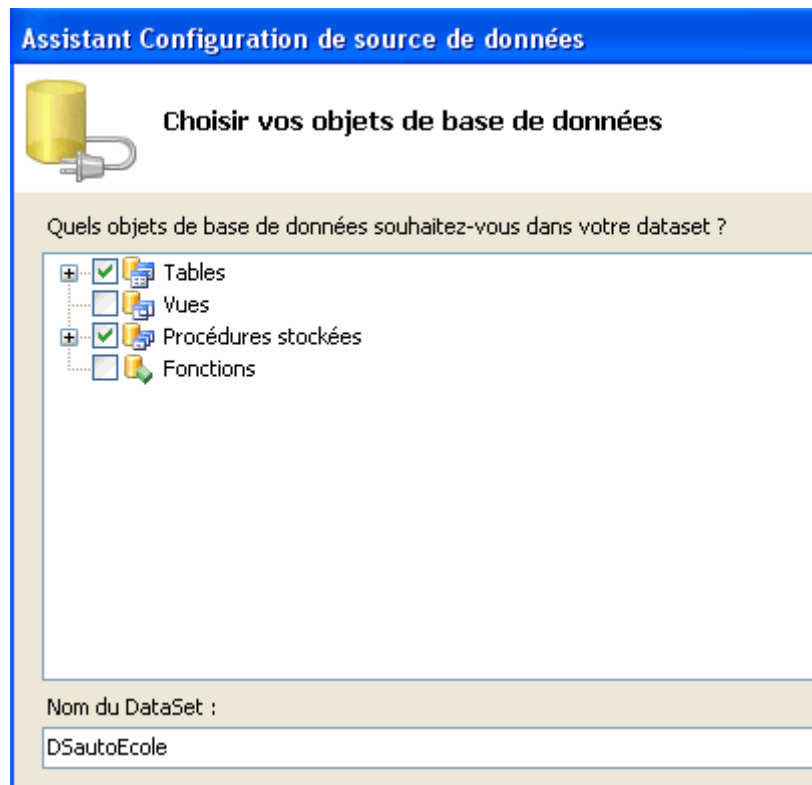


fig. 10 : création du DataSet

Le DataSet est généré ; on peut en voir une représentation :

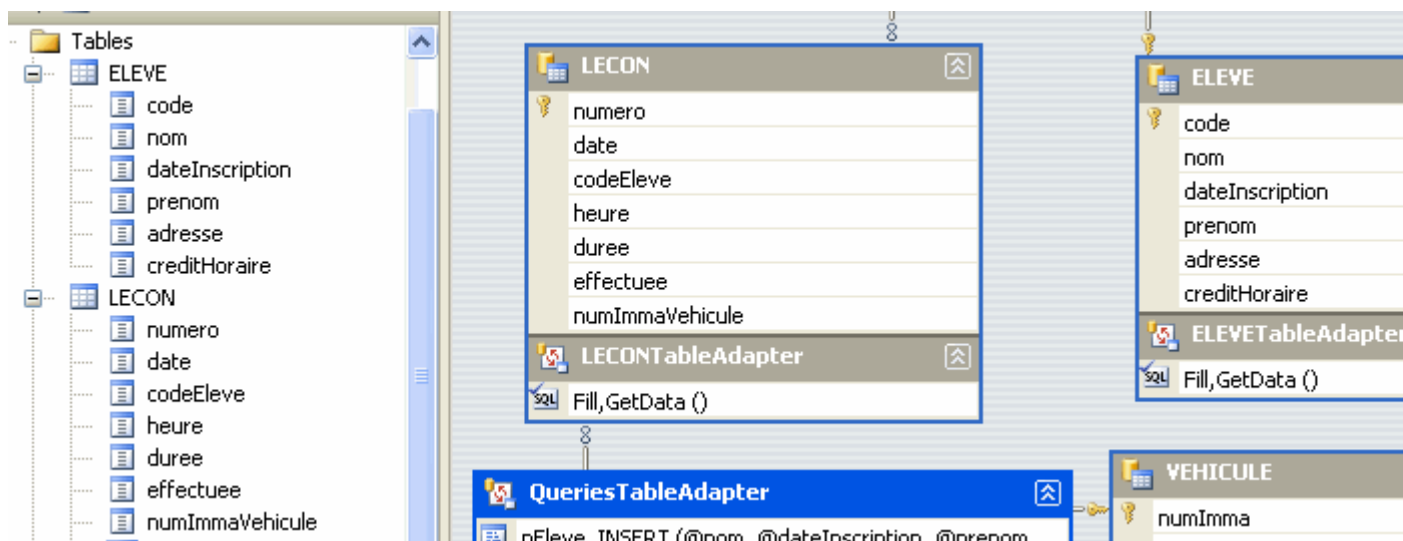


fig. 10 : composition du DataSet

On distingue :

- Dans la source de données (à gauche) la description tabulaire du DataSet
- La modélisation des classes : type **DataTable** pour les données et **TableAdapter** pour assurer la liaison avec la base de données
- Les procédures stockées sont interprétées comme une **QueriesTableAdapter**, avec une méthode pour la procédure.

- Le **DSautoEcole** a bien été ajouté au projet (partie droite). Cette classe hérite de la classe générique *DataSet*.

● La classe **DataSet**.

C'est la classe générique qui nous permet d'avoir une représentation tabulaires de données (une ou plusieurs tables, un fichier XML, etc...); c'est pourquoi elle est constituée d'une arborescence de collections de classes :

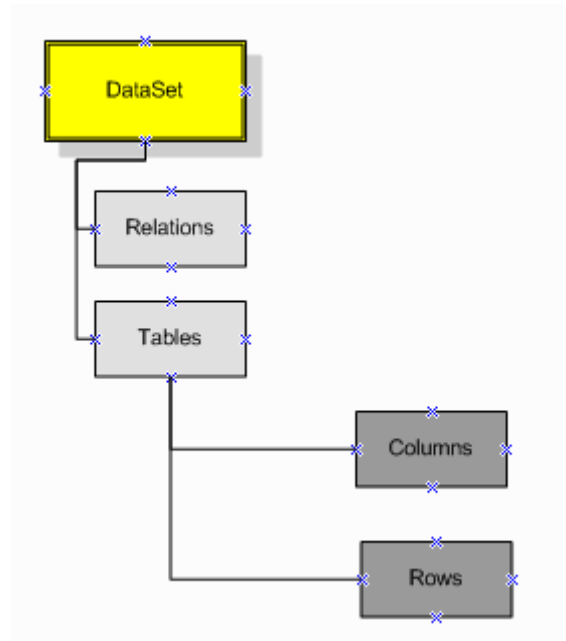


fig.11 extrait de la structure d'un DataSet

Le diagramme de classes suivant indique les relations entre les classes :

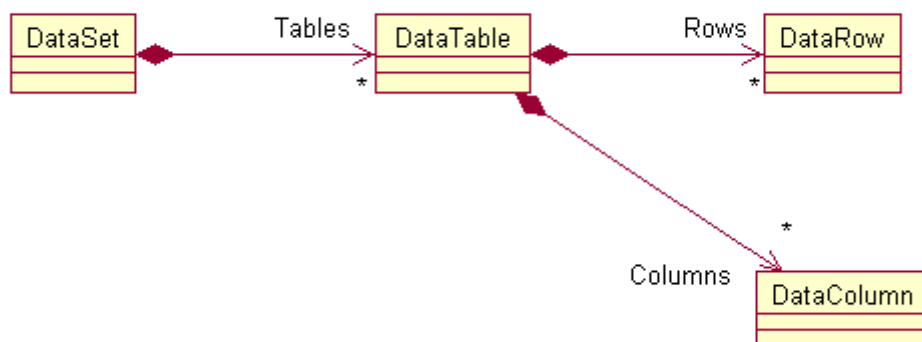


fig.12 diagramme de classe partiel

● Mise en oeuvre pour la gestion des véhicules

Nous créons un nouveau formulaire *FrmVehicule*, s'ouvrant lorsque l'on clique sur le *menu Véhicule* du formulaire d'accueil :

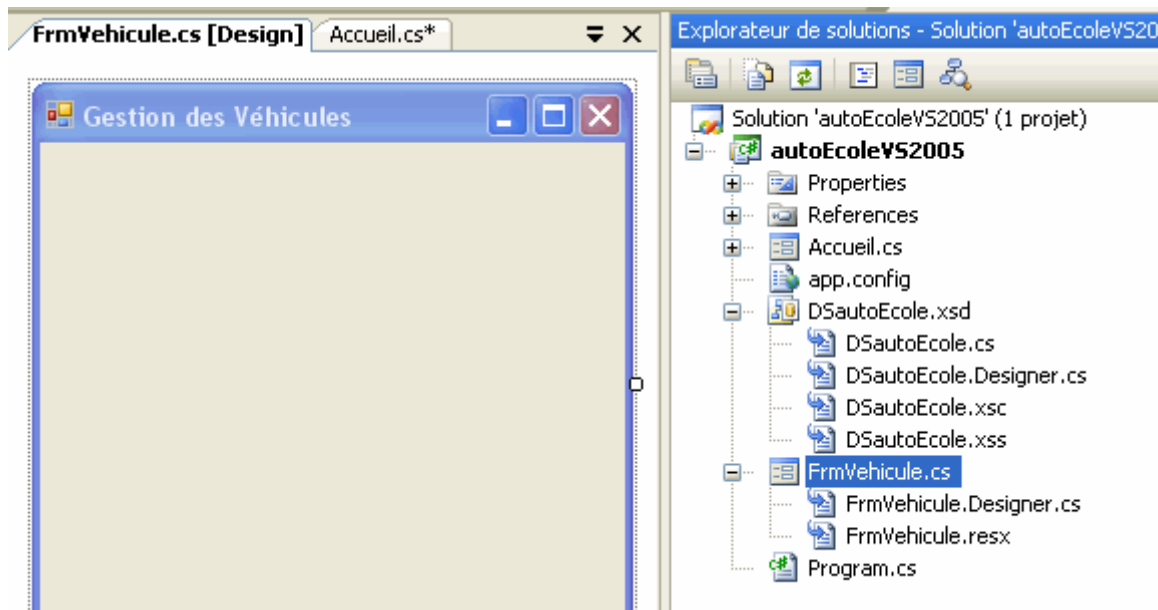


fig 13 : ajout d'un nouveau formulaire

```
private void menuGestion_Click(object sender, EventArgs e)
{
    FrmVehicule frmVehicule = new FrmVehicule();
    frmVehicule.Show();
}
```

Modifions l'interface en ajoutant les composants nécessaires à la gestion des véhicules :

The image shows a Windows form titled 'Gestion des Véhicules'. The form has a light beige background and a blue title bar. It contains three labels and three text input fields arranged vertically. The first label is 'Immatriculation' followed by a text box. The second label is 'Modèle' followed by a text box. The third label is 'Couleur' followed by a text box.

fig. 14 interface de gestion des véhicules

Nous allons associer maintenant chaque composant à une donnée. Ceci se fait en suivant plusieurs étapes :

Etape 1 Utilisation du dataSet

Déposer un DataSet sur le formulaire (Boîte à outils/Données/DataSet) :

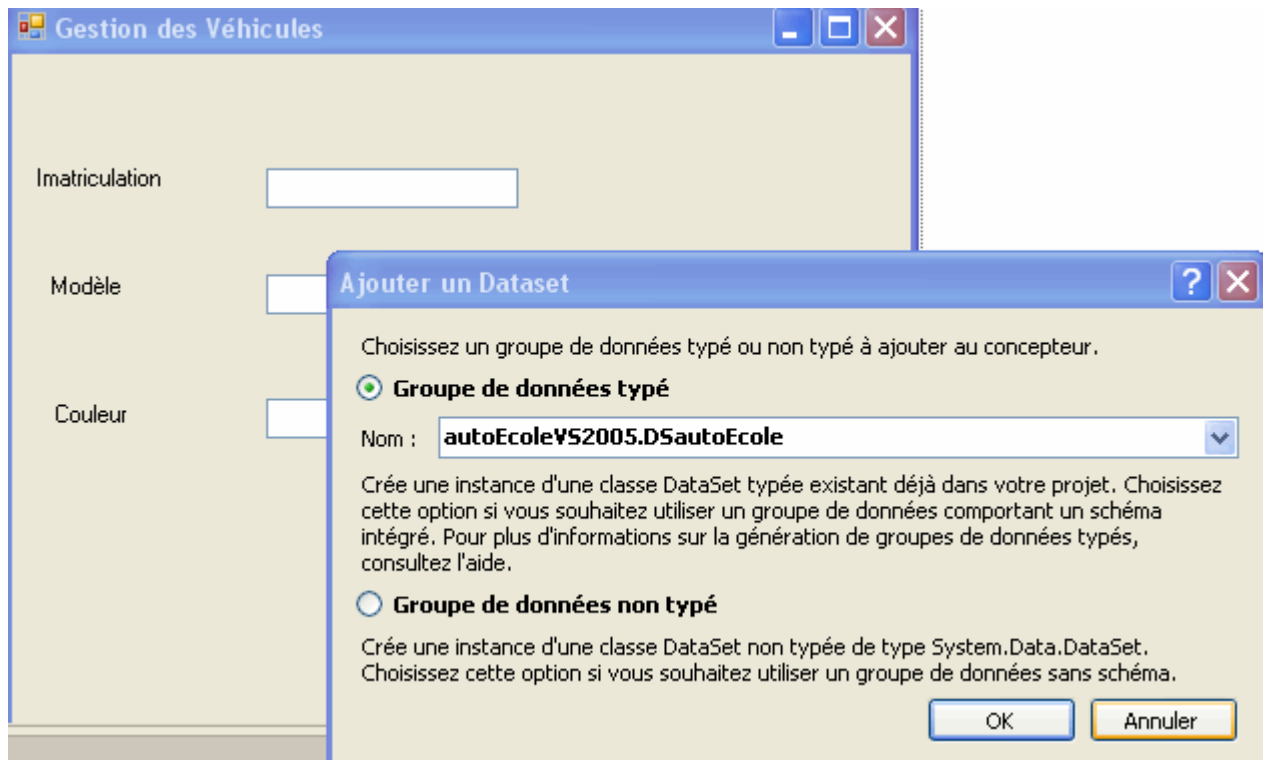


fig. 15 : sélection du DataSet

Visual Studio dépose une instance du DataSet :

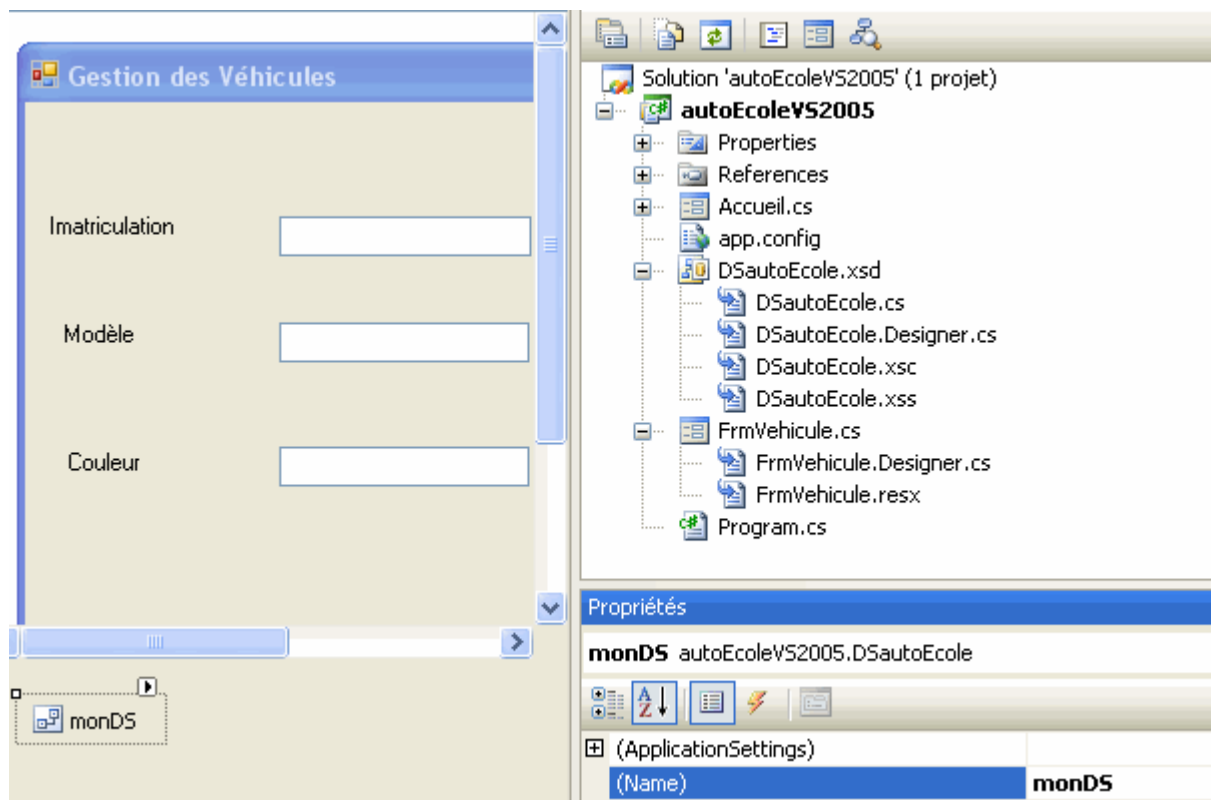


fig. 16 dépôt d'une instance de DataSet typé

Remarques :

- Nous avons renommé l'instance de DataSet : monDS
- Il s'agit bien d'un **objet de type DSautoEcole**

Lorsque l'on génère la solution (**F6**), la boîte à outil s'enrichit de nouveaux composants :

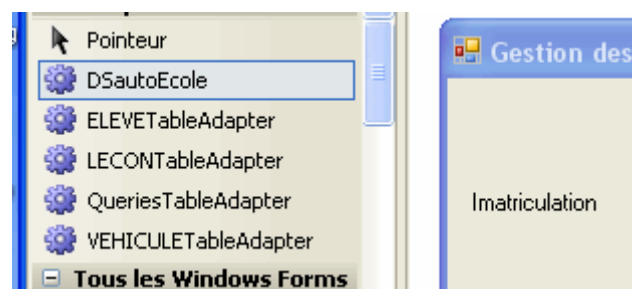


fig 17 création de nouveaux composants

Ce sont les tableAdapter (présentés plus haut) : un par table.

Ajoutons un **VehiculeTableAdapter** au formulaire :

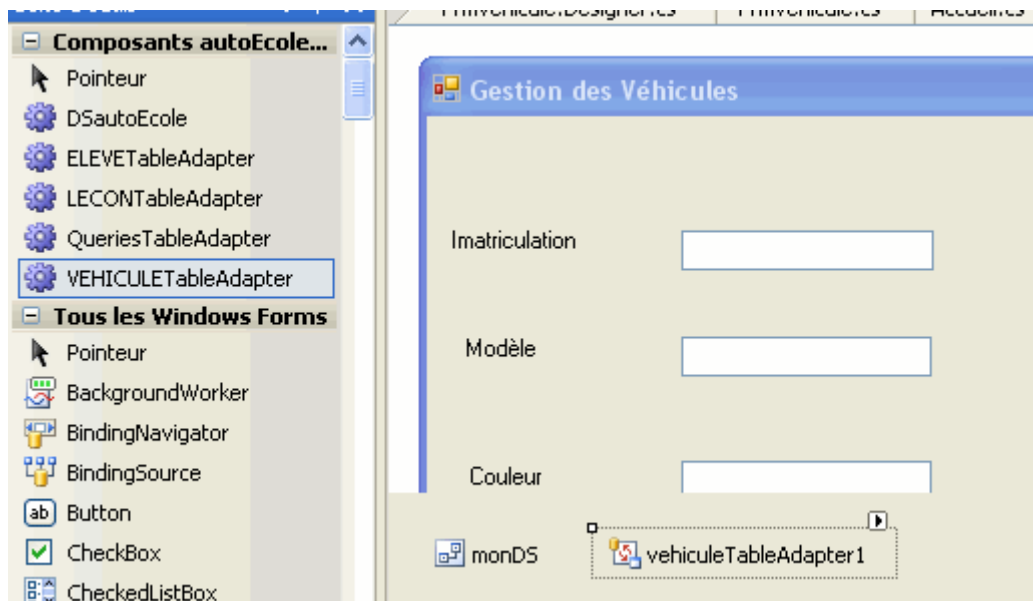


fig. 18 ajout d'un TableAdapter

■ Etape 2 Liaison des données avec les composants graphiques

Visual Studio propose un mécanisme très élaboré de liaison de données : le Binding ([Pour en savoir plus](#))

Le modèle suivant présente les responsabilités des différentes classes :

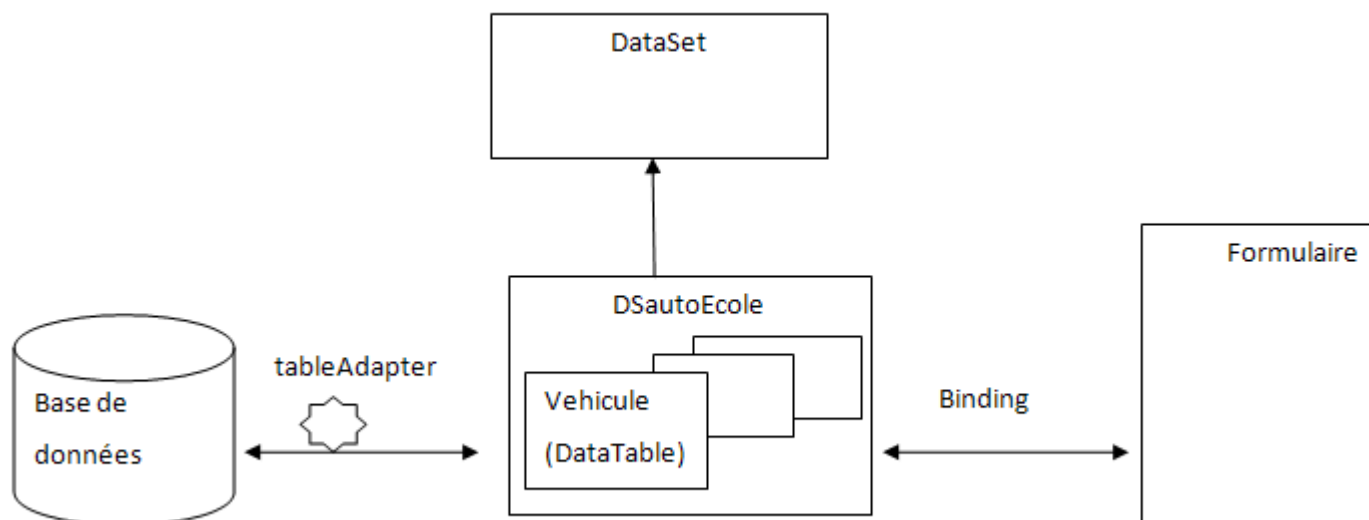


fig 19 : modèle global

Ainsi pour Binder le formulaire à la DataTable Vehicule, il faut ajouter un composant de binding et le lier à la Datatable.

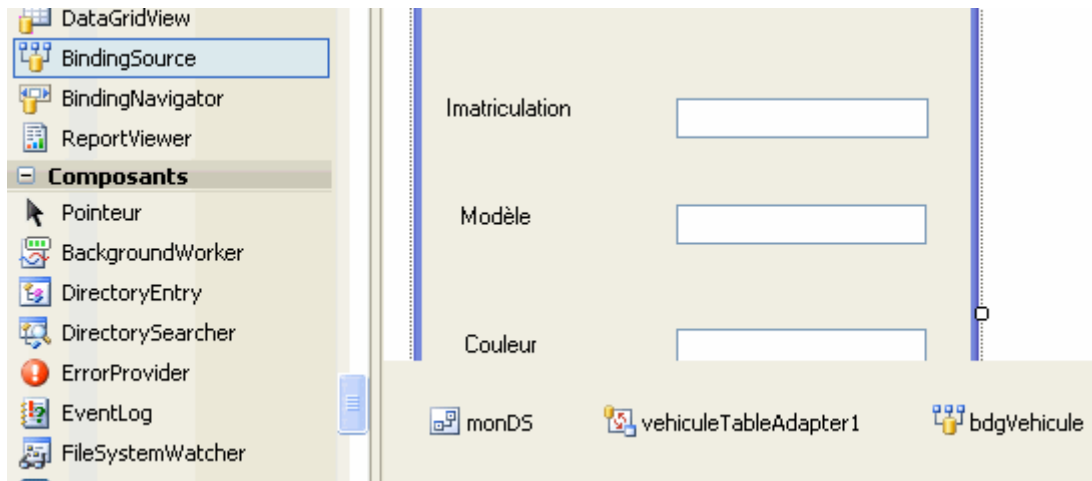


fig 20 : ajout d'un composant de Binding

Nous avons renommé ce composant : *bdgVehicule*

Configurons-le :

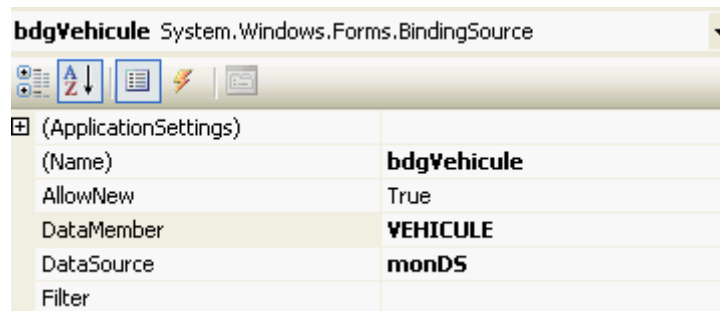


fig 21 : configuration du composant de Binding : liaison à la Datatable Vehicule du DSautoEcole

Cette opération lie au composant de Binding la DataTable VEHICULE

Maintenant, associons chaque composant graphique(les 3 zones de texte) au composant de Binding :

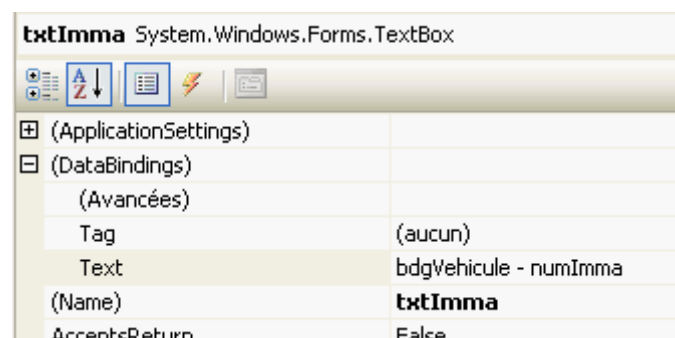


fig. 22 : la zone de texte txtImma (pour sa propriété Text) est liée au champ numImma du composant de Binding

Faisons ainsi pour les 2 autres zones de texte (modele et couleur).

Testons l'application : rien ne s'affiche !!

En effet, une action importante doit être menée : le chargement de la *DataTable*. Comme nous l'avons vu plus haut, c'est de la responsabilité d'un *TableAdapter* qui va remplir les lignes de la *DataTable* VEHICULE. Il faut écrire un peu de code, ceci peut se faire dans le constructeur du formulaire :

```
public FrmVehicule()  
{  
    InitializeComponent();  
    vehiculeTableAdapter1.Fill(monDS.VEHICULE);  
}
```

Si nous relançons l'application, nous voyons apparaître le premier véhicule :

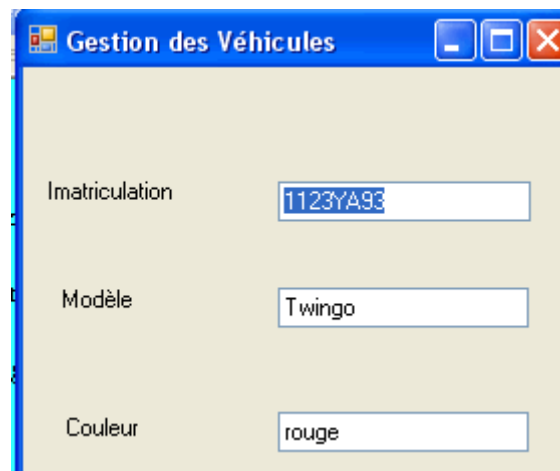


fig 23 : affichage du premier véhicule grâce au Binding

Mais, comment naviguer parmi les véhicules ? Visual Studio va aussi proposer un composant de navigation.

● Etape 3 Navigation dans une table

Nous allons utiliser un *BindingNavigator* pour parcourir la *DataTable* Véhicule.

Déposons ce nouveau composant dans le formulaire :

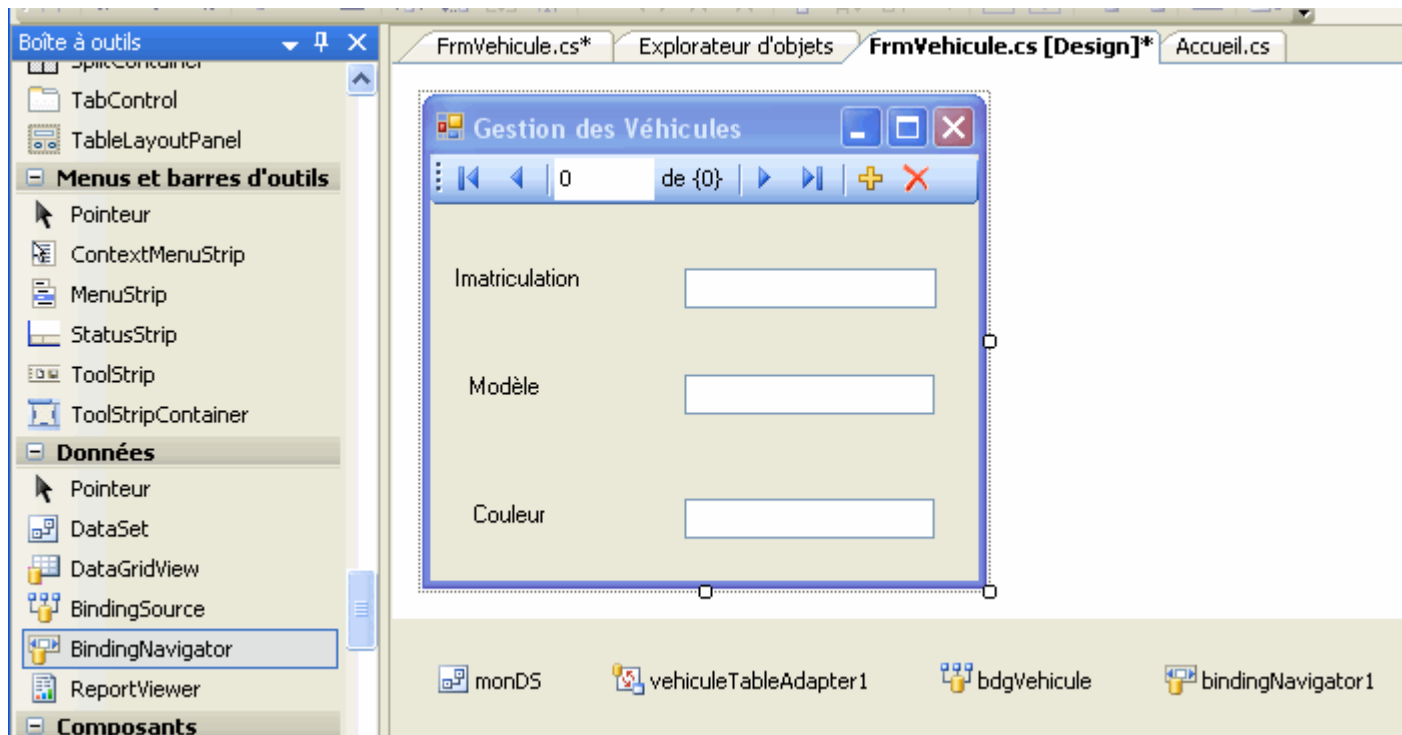


fig. 24 : dépôt d'un BindingNavigator dans le formulaire

Une barre de parcours apparaît en haut du formulaire. Il faut le configurer et le lier au composant de *Binding*.

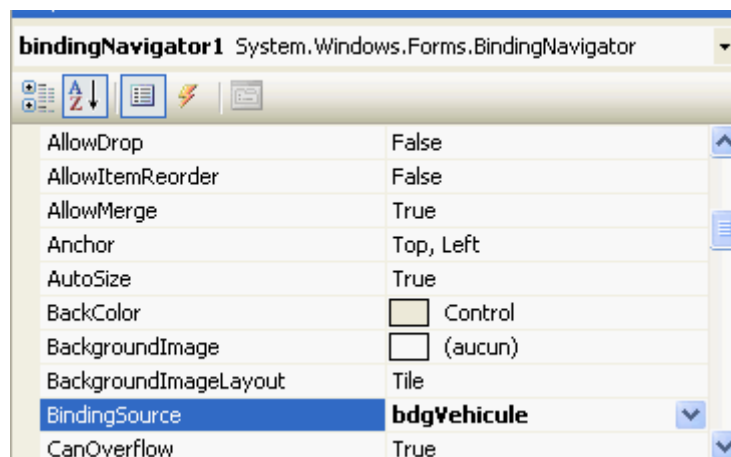


fig. 25 : configuration du BindingNavigator

Si nous lançons l'application, le formulaire se comporte de manière satisfaisante. Si nous essayons d'ajouter un nouveau véhicule celui-ci apparaît. Par contre si nous fermons et ouvrons le formulaire le véhicule disparaît. C'est bien sûr normal car les informations sont lues à partir du DataSet. Pour enregistrer les mises à jours dans la base de données, il faut demander au TableAdapter de le faire.

Ajoutons pour cela un nouveau bouton dans la barre de navigation, modifions l'image proposée -si nous en disposons- ou demander à afficher du texte (Sauver) ; faisons apparaître le gestionnaire d'événement sur le click du bouton, écrivons le code :

```
bdgVehicule.EndEdit();  
vehiculeTableAdapter1.Update(monDS.VEHICULE);
```

La première ligne permet de mettre à jour le DataSet, même si on a pas rafraîchi la saisie (cas où on demande l'enregistrement dans la base immédiatement après avoir fait la modification).

Tester l'application.

Plusieurs situations entraînent une erreur et une sortie violente du programme :

- saisie d'une nouvelle voiture avec un numéro d'immatriculation existant
- saisie d'une valeur nulle pour ce numéro
- suppression d'un véhicule pour lequel une leçon existe

Nous allons "gérer" l'erreur en utilisant un mécanisme de *gestion d'erreurs (ou gestion des exceptions)*.

La table "Vehicule" du DataSet intègre les contraintes du modèle relationnel, ici l'unicité de la valeur de la clé ou sa contrainte de champ non nul pour la clé . Si nous saisissons une valeur existante comme numéro d'immatriculation nous produisons une erreur d'exécution. Pour éviter cela nous allons mettre en oeuvre la gestion des exceptions proposée par C#. Une exception est une erreur générée lors de l'exécution du code. C# propose un mécanisme analogue à C ou java : **un bloc try** contient les instructions susceptibles de provoquer des erreurs, un **bloc catch** contient le code qui s'exécute lorsque une erreur (gérable) survient. ([Pour en savoir plus](#))

Les deux premières *exceptions* sont déclanchées au moment où on sort de la zone de saisie en actionnant un autre bouton.

Dans le gestionnaire d'événement du click sur un quelconque bouton du BindingNavigator, ajoutons le code :

```
private void bindingNavigator1_ItemClicked(object sender, ToolStripItemClickedEventArgs e)  
{  
    try  
    {  
        bdgVehicule.EndEdit();  
    }  
    catch (Exception ex)  
    {  
        MessageBox.Show(ex.Message);  
        bdgVehicule.CancelEdit();  
    }  
}
```

La dernière instruction rejette tout changement.

Remarque : en toute rigueur on pourrait retirer l'instruction `bdgVehicule.EndEdit();` placée au moment de la sauvegarde puisqu'elle est réalisée à chaque

Par contre la dernière *exception* est envoyée par la base de données au moment de la suppression, lors de la méthode *Update*. Il faut donc intercepter cette exception aussi. Le code de l'appel de *Update* devient :


```

try
{
    vehiculeTableAdapter1.Update(monDS.VEHICULE);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}

```

● Vision tabulaire de la table Véhicule : le DataGridView

Visual Studio propose différents contrôles permettant une visualisation des données, le **DataGridView** figure à une bonne place parmi ceux-ci.

Nous pouvons visualiser les véhicules sous forme tabulaire :



fig 25: un DataGridView pour visualiser les véhicules

Créer un nouveau formulaire (*FrmListeVehicules*) qui s'ouvrira à partir du formulaire d'accueil sur l'option du menu *Vehicule/Liste*.

Déposer un DataGridView dans le formulaire créé, ainsi qu'un bouton de sauvegarde et un autre d'annulation.

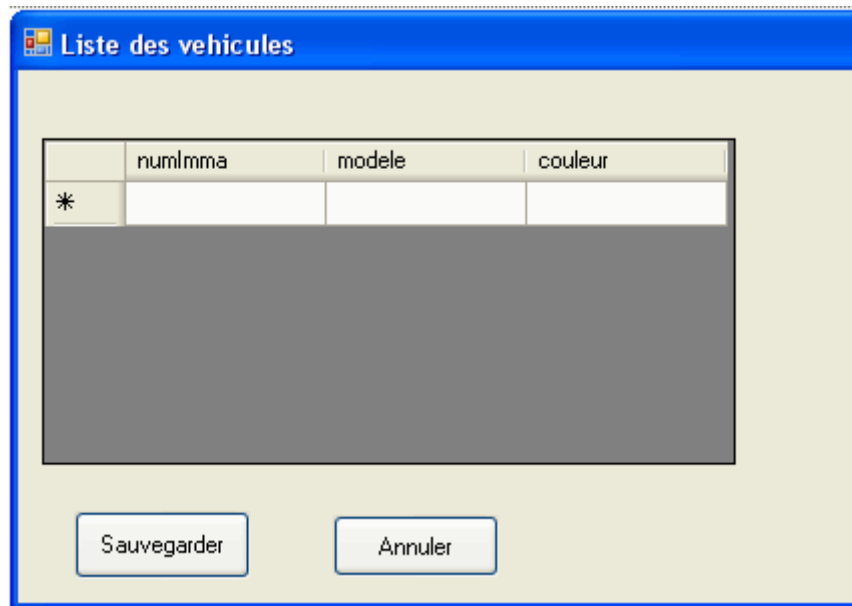


fig 26 formulaire en mode conception

Nous pouvons procéder comme pour le précédent formulaire et installer successivement :

- Le DataSet
- Le TableAdapter
- Le BindingSource.

Mais nous pouvons aussi demander à Visual Studio prendre en charge ces créations et configurations.

Pour cela, il suffit à partir du seul DataGridView, paramétrer sa propriété DataSource et indiquer que nous voulons le lier à la DataTable Vehicule du DataSet !!

Ce que nous allons faire. Nous observons deux choses.

- Visual Studio installe et configure les trois composants :

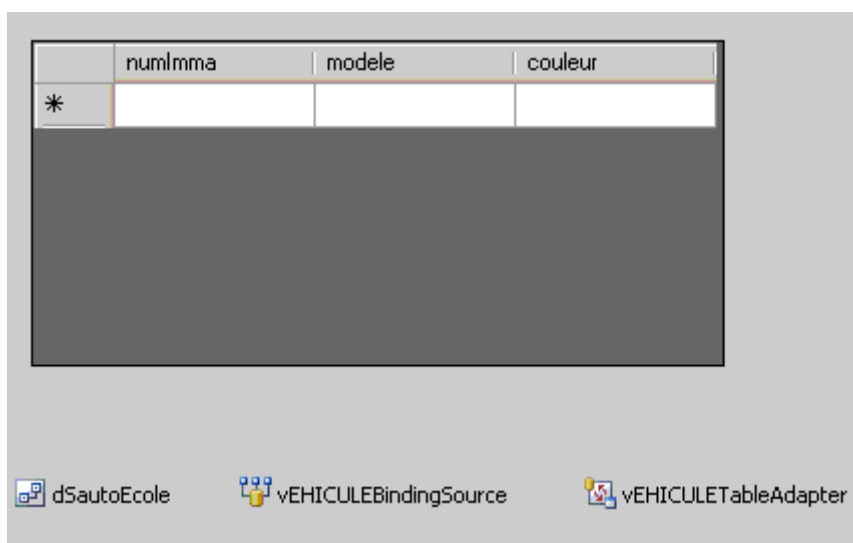


fig 27 : composants installés per visual Studio

- Visual Studio écrit la ligne de code qui charge le DataSet :

```
private void FrmListeVehicules_Load(object sender, EventArgs e)
{
    // TODO : cette ligne de code ....
    this.vEHICULETableAdapter.Fill(this.dSautoEcole.VEHICULE);
}
```

Gestion des exceptions :

Comme plus haut, nous interviendrons à tout événement click du DataGridView :

```
private void dataGridView1_Click(object sender, EventArgs e)
{
    try
    {
        vEHICULEBindingSource.EndEdit();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
        vEHICULEBindingSource.CancelEdit();
    }
}
```

Il faut cependant gérer la mise à jour de la base (méthode *Update*); c'est le rôle du bouton "Sauvegarder" :

```
private void btnSauve_Click(object sender, EventArgs e)
{
    try
    {
        this.vEHICULETableAdapter.Update(dSautoEcole.VEHICULE);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Le bouton "Annuler" rejette toutes les modifications :

```
private void btnAnnuler_Click(object sender, EventArgs e)
{
    dSautoEcole.VEHICULE.RejectChanges();
}
```

Remarque : c'est la **DataTable** qui rejette les mises à jour.

Gestion des élèves, utilisation de procédures stockées

Nous allons maintenant utiliser une autre technique pour ajouter un nouvel élève. Regardons le formulaire de création d'un nouvel élève :

fig 28 création d'un nouvel élève

Créons ce formulaire :

- La zone de forfait (ComboBox) va contenir des valeurs en "dur".
- La date d'inscription comporte une zone de texte et un composant **MonthCalendar**
- Pensons à renommer les composants qui seront utilisés dans le code : txtNom, txtPrenom, txtAdresse, cmbForfait, txtDate.

Il s'agit à partir de ce formulaire de créer un nouvel élève dans la base de données.

L'identifiant de l'élève est numérique, ce n'est pas à l'utilisateur de donner la valeur de ce nouvel identifiant (qui par ailleurs n'est pas signifiant). D'autre part dans un contexte multi-utilisateurs, nous n'aurions pas la garanti de l'unicité de cette valeur. Nous allons donc déporter la responsabilité de gestion de la valeur de l'identifiant à la base de données; ceci se fera sous la forme d'une procédure stockée. La procédure stockée aura ainsi deux responsabilités ; d'une part générer la valeur de l'identifiant et d'autre part mettre en oeuvre la requête d'insertion.

Avant cette étape, attachons nous à la gestion de l'interface :

- Le forfait horaire prendra des valeurs numériques 10, 15, 20, 25, 30, 35, 40 ; ces valeurs seront chargées dans le constructeur :

```
for (int i=5 ; i<40 ; i+=5)
    cmbForfait.Items.Add(i);
```

- Le TextBox "Date d'inscription" sera chargé à l'aide d'un contrôle **MonthCalendar** ; ce contrôle permet de récupérer facilement une date (ou une plage de dates). Pour limiter la sélection à une seule date il faut fixer la propriété MaxSelectionCount à 1. La valeur de la date sélectionnée est récupérée grâce à l'événement :

```
private void monthCalendar1_DateChanged(object sender, DateRangeEventArgs e)
{
    txtDate.Text = monthCalendar1.SelectionStart.ToShortDateString();
}
```

● Gestion de la procédure stockée.

Cette procédure est déjà installée dans la base de données.

Nous allons la visualiser grâce à l'explorateur de serveur" (Affichage/Explorateur de serveurs) :

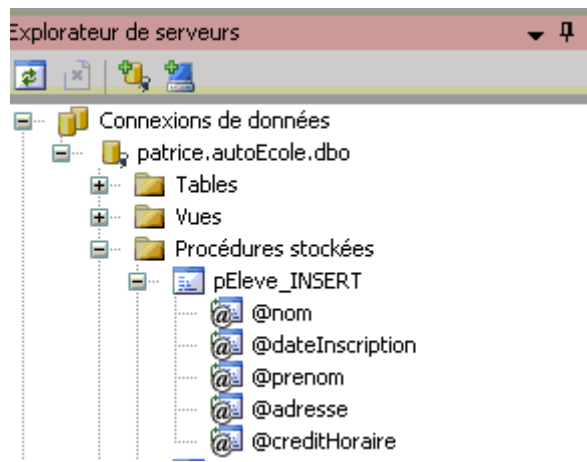


fig 29 parcours de l'explorateur de serveurs

Si nous ouvrons la procédure on peut observer son code :

```

dbo.pEleve_INS...rice.autoEcole)  FrmCreationEleve.cs [Design]  Accueil.cs
/***** Objet : Procédure stockée dbo.pEleve_INSERT */
ALTER PROC pEleve_INSERT
    @nom nvarchar(8)
    ,@dateInscription smallDateTime
    ,@prenom nvarchar(15)
    ,@adresse nvarchar(30)
    ,@creditHoraire int
AS
Declare @code smallint
SELECT @code = (select max(code) from eleve)
select @code = (@code+1)
INSERT eleve (
    code
    ,nom
    ,dateInscription
    ,prenom
    ,adresse
    ,creditHoraire
)
VALUES (
    @code
    ,@nom
    ,@dateInscription
    ,@prenom
    ,@adresse
    ,@creditHoraire
)

```

fig 30 : code de la procédure stockée d'insertion d'un nouvel élève

● Mise en oeuvre de la procédure stockée.

Un composant de données a été automatiquement généré à la création de la source de donnée (cf plus haut) ; c'est le **QueriesTableAdapter**, déposons-le dans le formulaire :

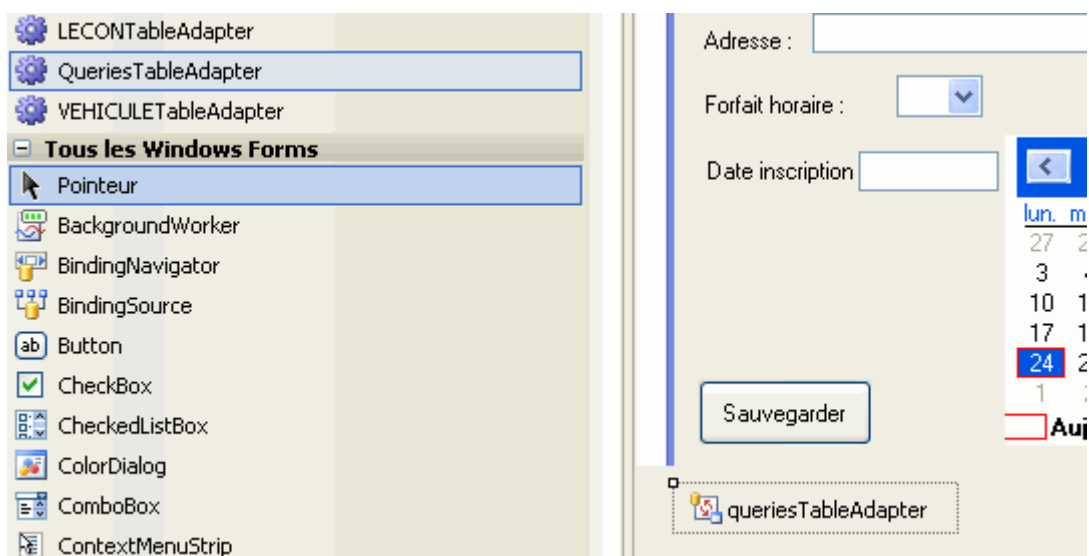


fig 31 : le composant QueriesTableAdapter

C'est ce composant qui a la responsabilité d'exécuter la procédure stockée ; dans le code de l'événement click du bouton Sauvegarder écrivons le code :

```
private void btnSauver_Click(object sender, EventArgs e)
{
    string nom = txtNom.Text ;
    DateTime? dt = Convert.ToDateTime( txtDate.Text);
    string prenom = txtPrenom.Text;
    int? forfait = Convert.ToInt32(cmbForfait.SelectedItem);
    string adresse = txtAdresse.Text;
    try
    {
        queriesTableAdapter.pEleve_INSERT(nom, dt, prenom, adresse, forfait);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Remarque : DateTime? et int? sont des types "nullables". *Un type nullable peut représenter la plage normale de valeurs pour son type valeur sous-jacent, **plus une valeur null supplémentaire** (in MSDN).* Ils sont utilisés ici car dans la base de données, les valeurs NULL sont autorisées.

Ne pas oublier de mettre ce code "sensible" dans un try/catch

Exécuter le programme et vérifier l'insertion d'un nouvel élève.

Création d'une leçon

Nous allons créer le formulaire de création de leçon. Ajouter un nouveau formulaire attaché à l'option *nouvelle leçon* du menu *Leçon*. Le formulaire doit se présenter ainsi :

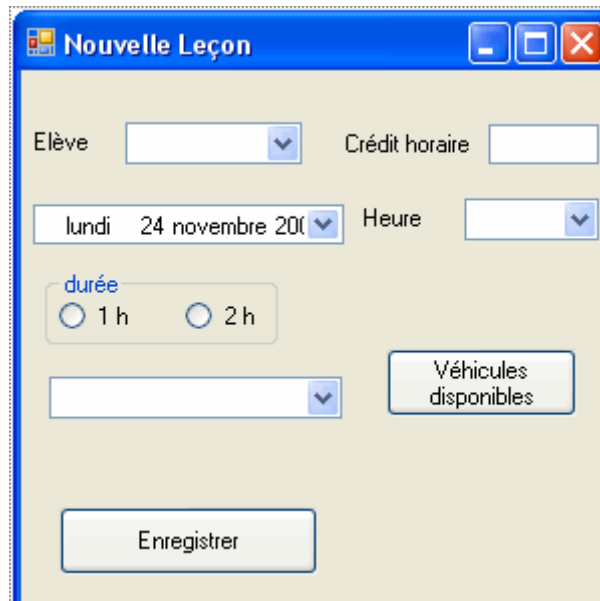


fig 32 saisie de leçon

- Un contrôle **DateTimePicker** a été placé pour saisir la date de la leçon
- Le comboBox donnera des valeurs en "dur" de 8h à 20h
- Lorsque l'on clique sur le bouton *Véhicules disponibles* le comboBox charge les véhicules disponibles à ce jour et cette heure.

● Gestion du ComboBox d'élèves et du crédit horaire.

Déposons un DataSet, un EleveTableAdapter et un composant de Binding, renommons-les :

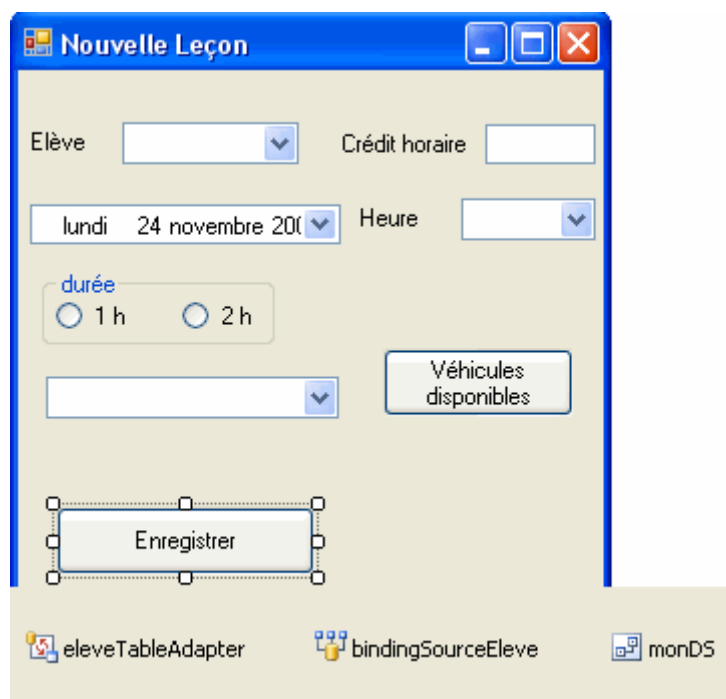


fig 32 : placement des composants pour la gestion de l'élève

Configurons le bindingSourceEleve :

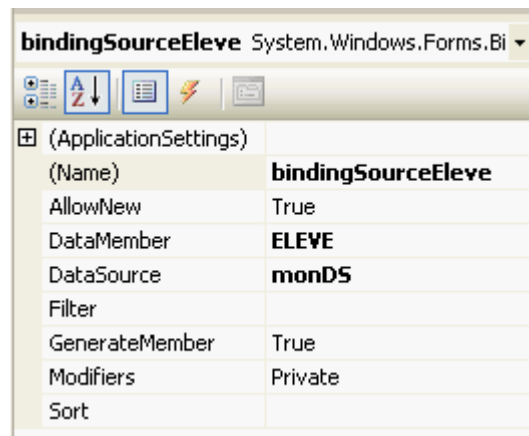


fig 33 : configuration du bindingSourceEleve

Lions le comboBox au composant de Binding :

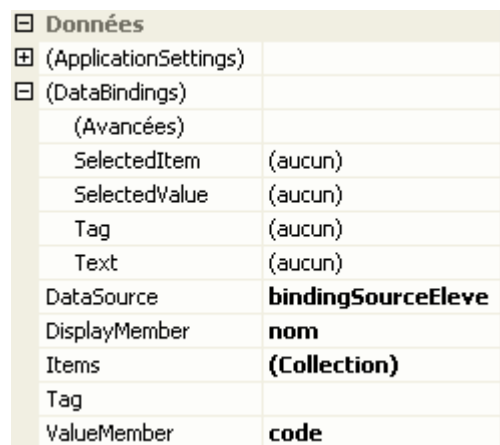


fig 34 : configuration ComboBox des élèves

Remarque :

- La propriété *DisplayMembre* fait référence au champ affiché
- La propriété *ValueMembre* est par contre le champ effectivement lié

La zone de texte des forfaits est également liée au champ *creditHoraire*

Si nous lançons l'application, après avoir bien sûr chargé le DataSet (méthode Fill du TableAdapter) :



fig 35 : test de la partie élève du formulaire

Le comboBox des horaires doit être chargé dans le constructeur du formulaire (de 8 à 20)

```
public FrmNouvelleLecon()  
{  
    InitializeComponent();  
    eleveTableAdapter.Fill(monDS.ELEVE);  
    for(int i=8;i<=20;i++)  
        cmbHeure.Items.Add(i);  
}
```

● Gestion du ComboBox des véhicules disponible : création d'une DataTable

Pour remplir le ComboBox des véhicules disponibles, nous allons créer une nouvelle DataTable.

Allons dans le schéma du DataSet (le fichier .xsd)

Ajoutons un nouveau TableAdapter, configurons-le :

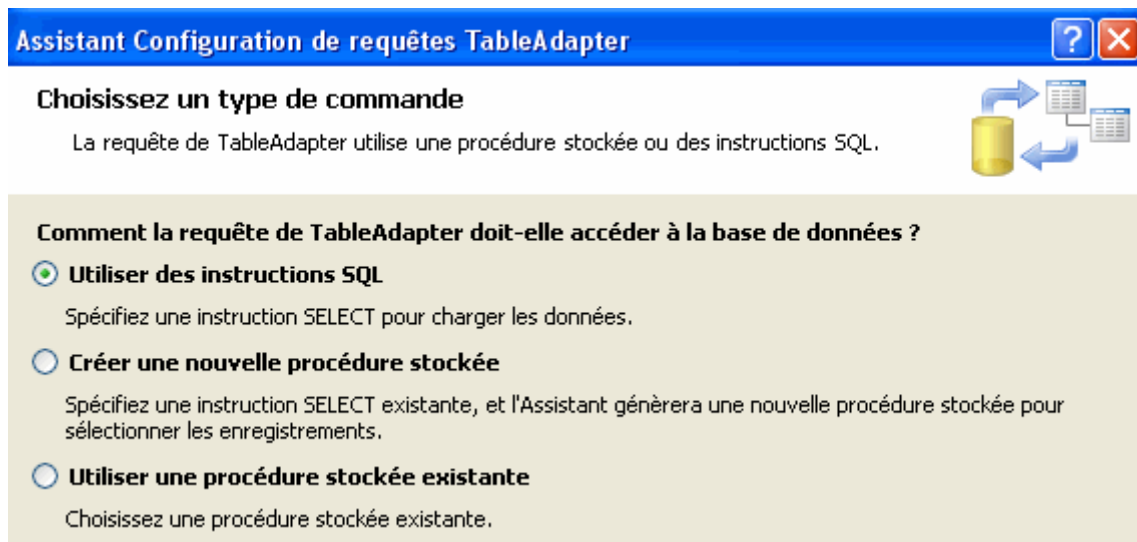


fig 36 : configuration du TableAdapter

Assistant Configuration de requêtes TableAdapter

Choisir un type de requête
Choisissez le type de requête à générer

Quel type de requête SQL voulez-vous utiliser ?

- ☒ **SELECT qui retourne des lignes**
Retourne une ou plusieurs lignes ou colonnes.
- ☐ **SELECT qui retourne une seule valeur**
Retourne une seule valeur (par exemple, Sum, Count ou une autre fonction d'agrégation).
- ☐ **UPDATE**
Modifie les données existantes dans une table.
- ☐ **DELETE**
Supprime des lignes d'une table.
- ☐ **INSERT**
Ajoute une nouvelle ligne à une table.

fig 37 : configuration du TableAdapter

Assistant Configuration de requêtes TableAdapter

Spécifier une instruction SQL SELECT
L'instruction SELECT sera utilisée par la requête.

Tapez votre instruction SQL ou utilisez le Générateur de requêtes pour la construire. Quelles données être chargées dans la table ?

Quelles sont les données que la table doit charger ?

```
select numImma from Vehicule where numImma not in
(select numImmaVehicule from Lecon where date = @dateLecon and heure=@heureLecon)
```

fig 38 : configuration du TableAdapter

Remarque : la requête est paramétrée par des valeurs qui seront saisies dans le formulaire (@dateLecon et @heureLecon)

Quelles méthodes voulez-vous ajouter au TableAdapter ?

☐ **Remplir un DataTable**
 Crée une méthode qui utilise un DataTable ou un Dataset comme paramètre et exécute l'instruction SQL ou la procédure stockée SELECT entrée dans la page précédente.

Nom de la méthode :

☒ **Retourner un DataTable**
 Crée une méthode qui retourne un nouveau DataTable rempli avec les résultats de l'instruction SQL ou de la procédure stockée SELECT entrée dans la page précédente.

Nom de la méthode :

☐ **Créer des méthodes pour envoyer directement des mises à jour à la base de données**
 Crée des méthodes Insert, Update et Delete qui peuvent être appelées pour l'envoi direct de modifications de lignes individuelles à la base de données.

fig 39 : configuration du TableAdapter

Terminer la configuration, nous pouvons voir la DataTable associée(renommée) et sa méthode GetData qui retourne une DataTable :

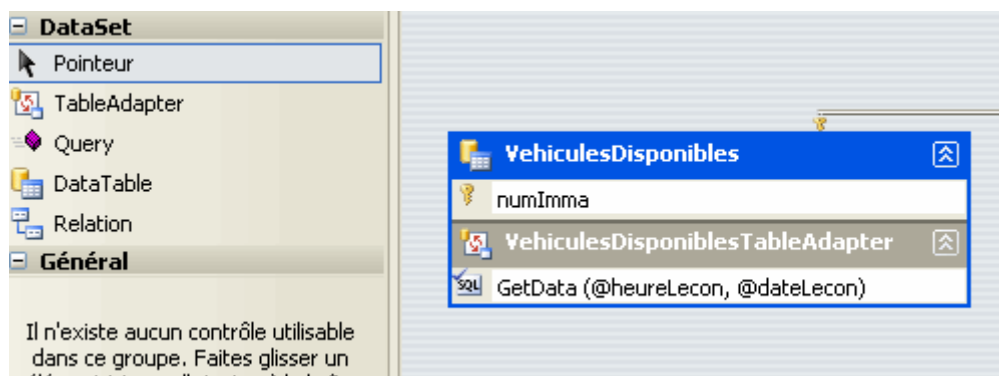


fig 40 : création de la DataTable

Retournons dans le formulaire, **regénérons le projet** ; on voit apparaître un nouveau composant de données :

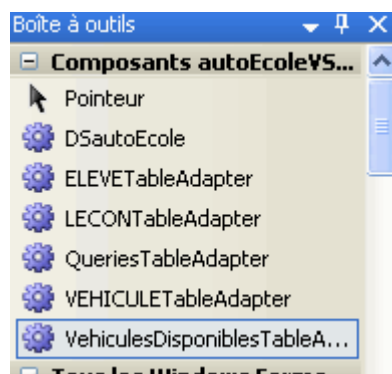


fig 41 : nouveau composant de données (TableAdapter)

Déposons ce composant dans le formulaire ; il ne nous reste plus qu'à charger le DataSet (méthode getData), lier (par le code) la DataTable au ComboBox :

```

private void btnVehicules_Click(object sender, EventArgs e)
{
    int? heure = Convert.ToInt32(cmbHeure.SelectedItem);
    DateTime dateLecon = dateTimePicker1.Value;
    DataTable dt = vehiculesDisponiblesTableAdapter1.GetData(heure, dateLecon);
    cmbVehicule.DataSource = dt;
    cmbVehicule.DisplayMember = dt.Columns[0].ColumnName;
}

```

Tester et vérifier que tout se passe bien.

● Enregistrement de la leçon

Nous allons procéder comme pour la création d'un élève : déporter cette responsabilité vers la base de données en créant une procédure stockée qui générera une nouvelle valeur de l'identifiant et insèrera la nouvelle leçon dans la table.

Ajouter une nouvelle procédure stockée en ouvrant l'explorateur de serveurs :

```

/***** Objet : Procédure stockée dbo.pLECON_INSERT */
CREATE PROC pLECON_INSERT
    @date smallDateTime
    ,@codeEleve smallint
    ,@heure smallint
    ,@duree smallint
    ,@effectuee bit
    ,@numImmaVehicule nvarchar(8)
AS
Declare @numero smallint
    SELECT @numero = (select max(numero) from lecon)
    select @numero = (@numero+1)
    INSERT lecon (
        numero
        ,date
        ,codeEleve
        ,heure
        ,duree
        ,effectuee
        ,numImmaVehicule
    )
    VALUES (
        @numero
        ,@date
        ,@codeEleve
        ,@heure
        ,@duree
        ,@effectuee
        ,@numImmaVehicule
    )

```

Dans le schéma du DataSet (fichier .xsd) ajoutons la procédure stockée au composant QueriesTableAdapter, afin d'obtenir :

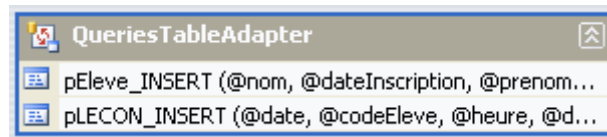


fig 42 : nouvelle procédure stockée intégrée

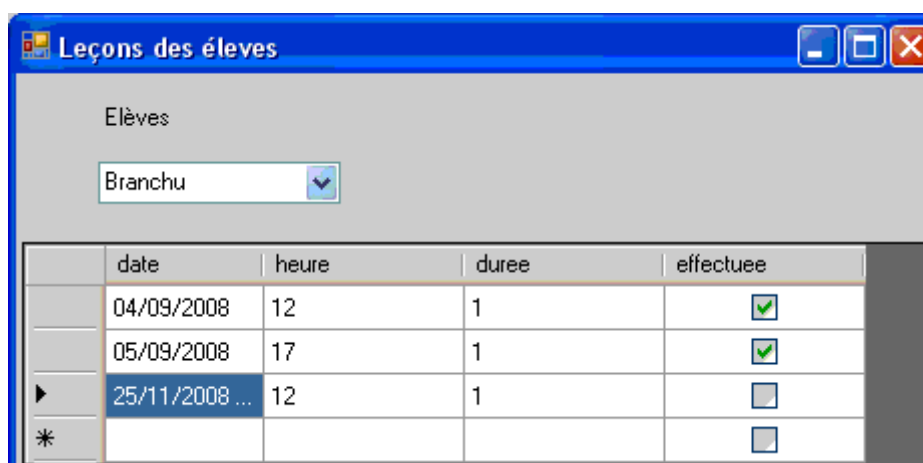
Dans le formulaire, déposons un composant QueriesTableAdapter ; terminons par le code :

```
private void btnEnregistrer_Click(object sender, EventArgs e)
{
    short? codeEleve = Convert.ToInt16(cmbEleve.SelectedValue);
    DateTime? date = dateTimePicker1.Value;
    short? heure = Convert.ToInt16(cmbHeure.Text);
    short? duree;
    if (radioButton1.Checked)
        duree = 1;
    else
        duree = 2;
    bool? Effectuee = false;
    string numImma = cmbVehicule.Text;
    queriesTableAdapter1.pLECON_INSERT(date, codeEleve, heure, duree, Effectuee,
numImma);
}
```

Tester l'insertion de nouvelles leçons.

Visualisation des leçons d'un élève : utilisation d'un trigger

Nous voulons obtenir le formulaire suivant :



	date	heure	duree	effectuee
	04/09/2008	12	1	<input checked="" type="checkbox"/>
	05/09/2008	17	1	<input checked="" type="checkbox"/>
▶	25/11/2008 ...	12	1	<input type="checkbox"/>
*				<input type="checkbox"/>

fig 43 leçons prises ou planifiées pour un élève

Créer un nouveau formulaire, appeler ce nouveau formulaire à partir du click sur le sous-menu *Eleve/Leçons*.

Ce formulaire permet de visualiser les leçons de chaque élève ainsi que de valider une leçon planifiée (champ *effectué* coché).

● Gestion du ComboBox des élèves

C'est la même opération que pour le formulaire précédent. Déposer un Dataset, un TableAdapterEleve, un composant de Binding.

Déposer un ComboBox, renseignez la propriété *DataSource*, la propriété *DisplayMembre* avec le champ nom et la propriété *ValueMember* avec le champ code. Appelons la méthode *Fill* dans le constructeur du Formulaire.

● Gestion du DataGridView

Nous sommes en présence, cette fois d'un composant graphique (le DataGridView) lié à un autre composant (le comboBox des élèves) ; nous ne voulons voir apparaître que les leçons d'un élève -celui qui est sélectionné-.

Le mécanisme à mettre en place est un peu différent.

Déposons un composant LeconTableAdapter qui chargera le DataSet

Déposons un DataGridView dans le formulaire, configurons la propriété DataSource :

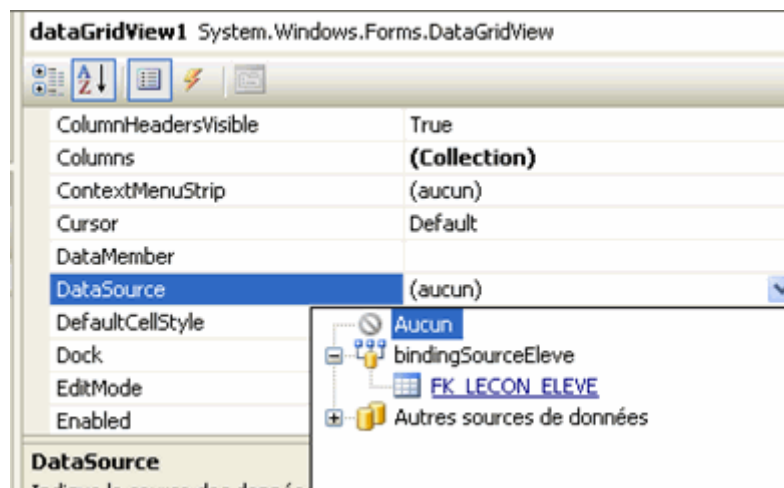


fig 44 : configuration du DataGridView

Le DataGridView sera "**bindé**" à la clé étrangère du composant de binding lié à la table ELEVE.

Lorsque nous validons la propriété, nous voyons apparaître un nouveau composant de Binding déposé par l'environnement.

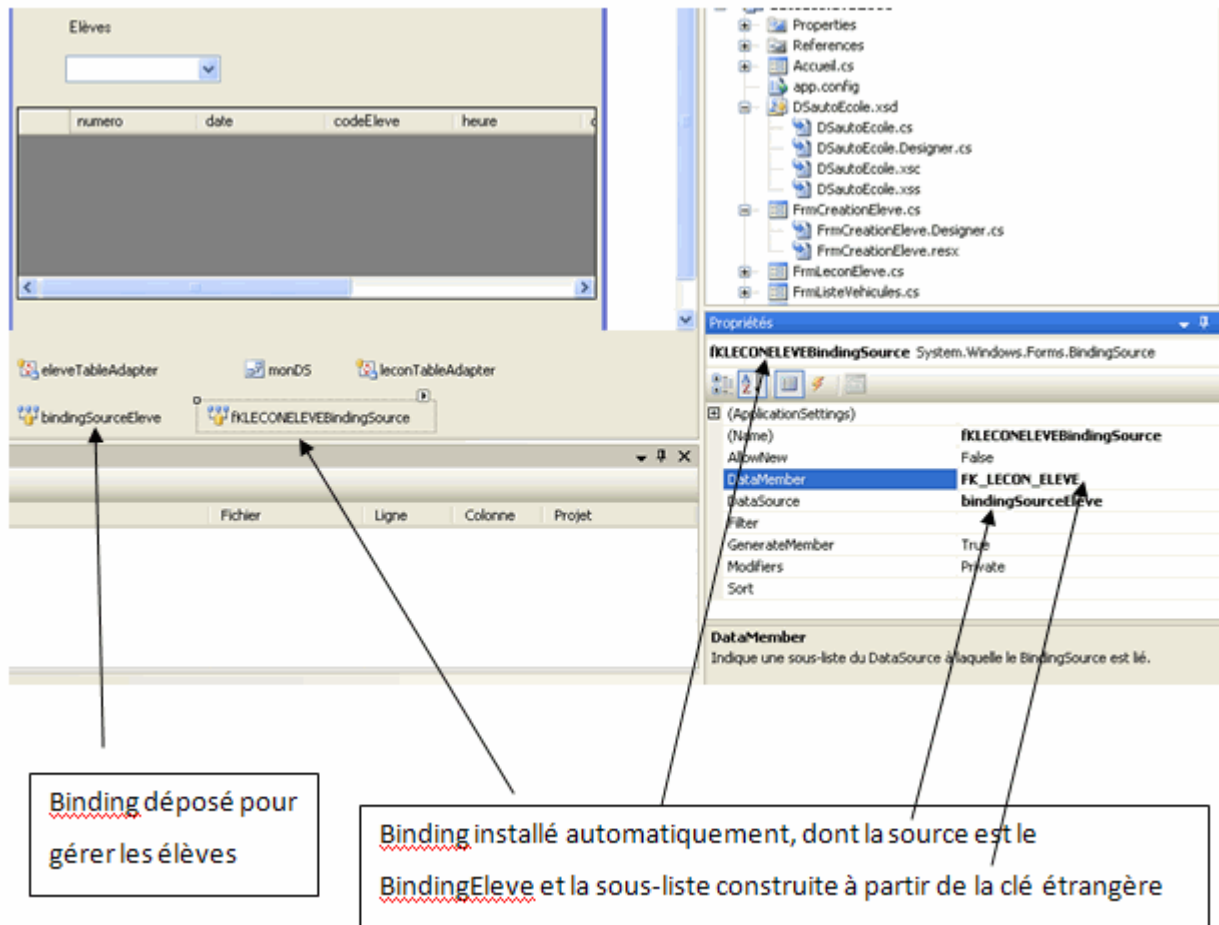


fig 45 : configuration du binding lié

Avant de tester, n'oublions pas d'appeler la méthode Fill du composant LeconTableAdapter :

```
public FrmLeconEleve()
{
    InitializeComponent();
    eleveTableAdapter.Fill(monDS.ELEVE);
    leconTableAdapter.Fill(monDS.LECON);
}
```

Il est possible de sélectionner les colonnes visibles du DataGridView : c'est sur la propriété Columns qu'il faudra intervenir :

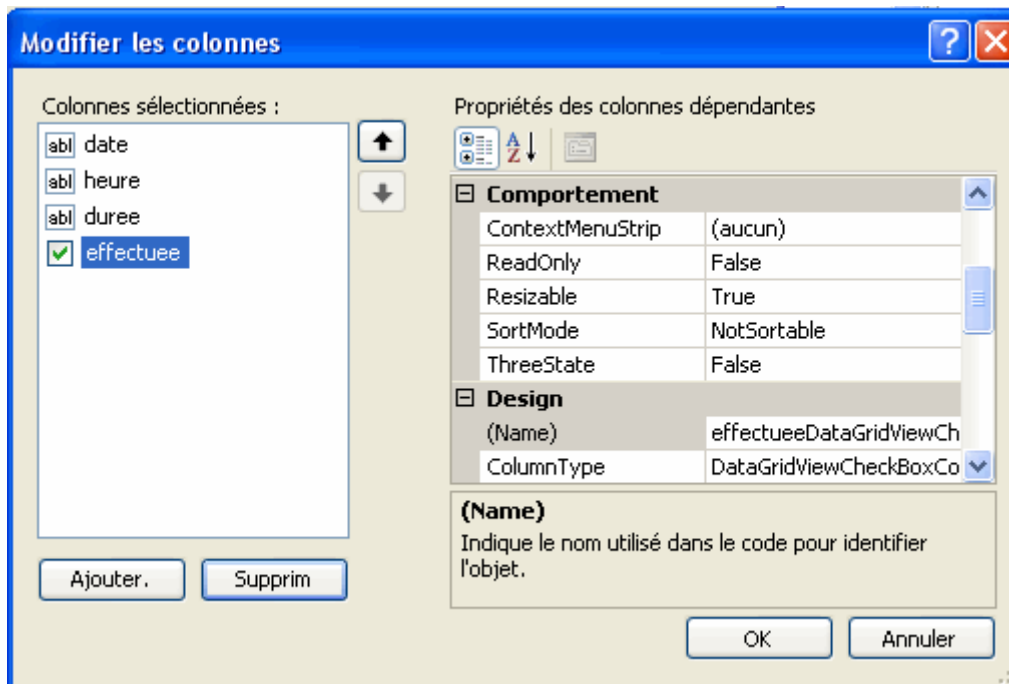


fig 46 : sélection des colonnes du DataGridView à faire apparaître

Tester.

● Enregistrement dans la base, trigger

L'écran sert à visualiser et aussi à valider le champ *effectuee* (case à cocher) de la table Leçon. Dans ce dernier cas il faut décrémenter les crédit horaire de l'élève concerné. Ceci peut se faire à l'aide d'un **trigger** (déclencheur) sur une clause **update** de la table Leçon.

Pour créer le trigger, utiliser l'explorateur de serveurs et ajouter un déclencheur sur la table Leçon.

```
CREATE TRIGGER LECON_Trigger1
ON dbo.LECON
FOR UPDATE
AS
    declare @codeEleve smallint,
            @duree smallint,
            @Oeffectuee bit,
            @Neffectuee bit
    select @codeEleve = codeEleve from inserted
    select @duree = duree FROM inserted
    select @Oeffectuee = effectuee FROM deleted
    select @Neffectuee = effectuee FROM inserted
    IF @Oeffectuee <> @Neffectuee
    BEGIN
        IF @Neffectuee = 1
        BEGIN
            UPDATE Eleve Set creditHoraire = creditHoraire - @duree
            WHERE code = @codeEleve
        END
    END
```

```

ELSE
BEGIN
    UPDATE Eleve Set creditHoraire = creditHoraire + @duree
    WHERE code = @codeEleve
END
END

```

Ce trigger se déclenchera à chaque modification de la table Leçon, testera si le champ effectuée a été modifié, si c'est le cas dans quel sens il l'a été et modifiera en conséquence le crédit horaire de l'élève concerné.

Pour valider les modifications, ajoutons dans le formulaire un bouton "Enregistrer"

```

private void btnEnregistrer_Click(object sender, System.EventArgs e)
{
    try
    {
        leconTableAdapter.Update(monDS.LECON);
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

Tester, valider une leçon et vérifier que le crédit horaire a bien diminué.