

# Gestion d'une base de données MySQL

## Avec les composants natifs de Lazarus

Par Alcatîz 

Date de publication : 29 août 2017

Dernière mise à jour : 9 septembre 2017

DÉBUTANT

Dans ce tutoriel, vous apprendrez à gérer une base de données MySQL en utilisant les composants natifs de Lazarus (SQLdb).

Trois exemples d'applications, de complexité croissante, vous sont proposés. Ils vont de l'utilisation exclusive de contrôles spécialisés en bases de données à celle de contrôles classiques. Quelques petits exercices (facultatifs) vous mettront au défi.

**Commentez**

I - Introduction.....	4
I-A - Outils de test.....	4
II - Création de la base de données de test.....	4
II-A - Création de la base.....	4
II-B - Création d'un utilisateur.....	5
II-C - Création des tables.....	6
II-D - Création des données.....	8
III - Exemple 1 : affichage simple du contenu d'une table.....	9
III-A - TMySQLConnection.....	9
III-B - Quel connecteur pour MariaDB ?.....	9
III-C - TSQLTransaction.....	10
III-D - TSQLQuery.....	10
III-E - TDataSource.....	10
III-F - TDBGrid.....	10
III-G - Commande d'affichage d'une table.....	11
III-H - Fermeture propre de l'application.....	11
III-I - Compilation et exécution.....	12
III-J - Code complet de l'exemple 1.....	12
IV - Exemple 2 : édition d'une table au choix.....	13
IV-A - Mot de passe de connexion.....	14
IV-A-1 - Affichage de l'erreur renvoyée par le système.....	15
IV-B - Récupération de la liste des tables.....	17
IV-C - Affichage du contenu de la table sélectionnée.....	18
IV-D - Permettre de modifier les données.....	19
IV-E - Code complet de l'exemple 2.....	21
V - Exemple 3 : une application complète.....	23
V-A - Création du projet.....	23
V-B - Unité de type DataModule.....	23
V-C - Utilisation des composants spécialisés.....	29
V-C-1 - TDBGrid.....	30
V-C-2 - TDBNavigator.....	32
V-C-3 - TDBEdit.....	32
V-C-4 - TDBRadioGroup.....	33
V-C-5 - Finalisation de la fiche.....	34
V-C-6 - Méthodes de chargement et de sauvegarde des données dans le datamodule.....	37
V-C-7 - Test de la fiche.....	38
V-C-8 - Exercice : réaliser le dialogue de gestion des clients.....	40
V-D - Une interface pour définir les requêtes SQL.....	40
V-E - Utilisation d'un TDBGrid sans TDBNavigator.....	43
V-E-1 - TDBGrid.....	43
V-E-2 - Filtres.....	45
V-E-3 - Requête SQL de sélection.....	46
V-E-4 - Chargement du DBGrid.....	47
V-E-5 - Exercice : ajouter un filtre pour n'afficher que les locations en cours.....	49
V-E-6 - Des boutons classiques pour l'ajout, la modification et la suppression.....	50
V-F - Utilisation de composants non spécialisés.....	50
V-F-1 - Chargement des clients et des voitures dans des TComboBox.....	52
V-F-2 - Requêtes de sélection des voitures et des clients.....	54
V-F-3 - Indices des client et voiture courants.....	56
V-F-4 - Estimation du prix de la location.....	58
V-F-5 - Réaction aux modifications des filtres.....	60
V-F-6 - Test de disponibilité de la voiture.....	60
V-F-7 - Enregistrement de la location.....	61
V-F-8 - Une classe descendante pour le dialogue de modification de location.....	64
V-F-8-a - Initialisation des champs.....	65
V-F-8-b - Enregistrement de la location modifiée.....	67
V-F-9 - Retour d'une voiture louée.....	70
V-F-10 - Exercice : supprimer une location.....	70

V-G - Toujours plus loin : une facture avec LazReport.....	71
V-G-1 - Requête de sélection.....	71
V-G-2 - Composants LazReport.....	71
V-G-3 - Conception du rapport.....	72
V-G-3-a - Données statiques.....	75
V-G-3-b - Données calculées.....	76
V-G-3-c - Données automatiques.....	78
V-G-4 - Code de création de la facture.....	78
V-H - Code complet de l'exemple 3.....	80
VI - Conclusion.....	80
VI-A - Remerciements.....	80

## I - Introduction

**MySQL** est un système de gestion de bases de données performant très largement utilisé. Il est actuellement la propriété de la société **Oracle** ; sa déclinaison communautaire **MariaDB**, sous licence GPL, est de plus en plus répandue dans le monde du libre.

Cet article est sans prétention ; son but est juste de vous guider dans la réalisation de vos premières applications utilisant une base de données MySQL, sous Lazarus.

Comme illustration, nous allons simuler la gestion d'une petite société de location de voitures.


### I-A - Outils de test

Si vous voulez installer rapidement un petit serveur vous permettant de réaliser les applications de ce tutoriel, vous pouvez opter pour une solution tout-en-un, par exemple :

-  **WAMP sous Windows** ;
-  **LAMP sous Linux**.

## II - Création de la base de données de test

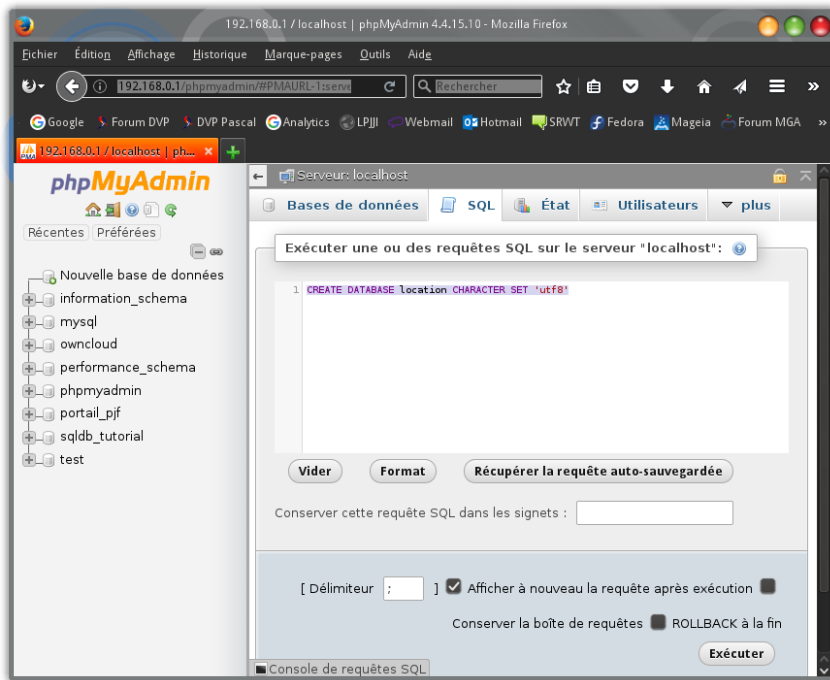
Avant toute chose, créons la base de données qui servira dans le cadre de ce tutoriel.

J'ai pris le parti d'utiliser l'interface de gestion  **phpMyAdmin**, qui permet de visualiser très facilement le résultat des actions que nous entreprendrons. Pour vous simplifier les choses, vous pouvez coller directement les commandes SQL dans l'onglet **SQL** de phpMyAdmin ; ce n'est toutefois pas une obligation et vous pouvez aussi exécuter à la main les différentes actions par le biais de l'interface.

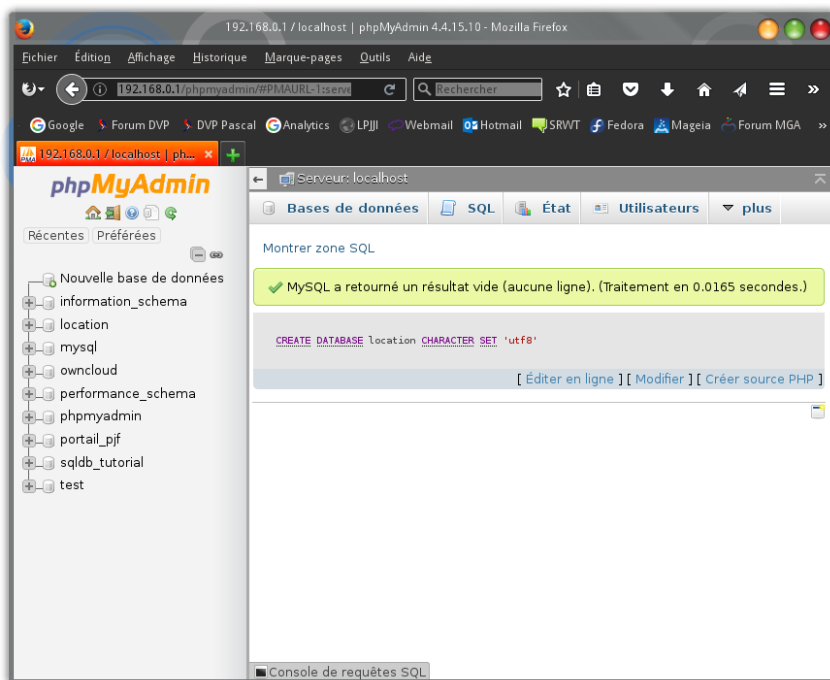
### II-A - Création de la base

Commençons donc par créer notre base de données. Dans l'onglet **SQL**, recopiez ou collez la commande suivante :

```
CREATE DATABASE location CHARACTER SET 'utf8';
```



Cliquez sur [Exécuter](#) et vérifiez que la nouvelle base de données est bien créée :



Le message de réussite sur fond vert et l'ajout de l'entrée [location](#) dans la colonne de gauche attestent la réussite de l'opération.

## II-B - Création d'un utilisateur

Il n'est jamais conseillé de travailler, dans quel système que ce soit, sauf lorsque c'est indispensable, avec des privilèges d'administrateur. C'est le cas avec MySQL et la première chose que nous allons faire est de créer un utilisateur qui aura spécifiquement accès à notre base de données, et à aucune autre.

Collez la commande suivante :

```
CREATE USER 'mysqldvp'@'%' IDENTIFIED BY 'passmysqldvp';GRANT ALL PRIVILEGES ON
location.* TO 'mysqldvp'@'%' IDENTIFIED BY 'passmysqldvp' REQUIRE NONE WITH GRANT OPTION;
```

Par cette commande, l'utilisateur [mysqldvp](#) a été créé et il a accès à la base de données [location](#) et à ses tables moyennant le mot de passe [passmysqldvp](#). Vous pouvez bien sûr définir un autre nom d'utilisateur et un autre mot de passe.

Si votre serveur MySQL tourne sur votre machine, vous pouvez remplacer '[mysqldvp](#)'@'%' par '[mysqldvp](#)'@'localhost'.

## II-C - Création des tables

Dans la liste des bases de données, à gauche, cliquez sur [location](#) ou collez la commande suivante dans l'onglet [SQL](#) :

```
USE location;
```

Dans l'onglet [SQL](#) de [location](#), nous allons créer des tables dans la base de données.

Nous avons besoin de trois tables :

- la liste des voitures ;
- le répertoire des clients ;
- la liste des locations.

La liste des voitures contiendra les colonnes suivantes :

Nom de colonne	Type	Commentaires
Plaque	VARCHAR(12)	Plaque d'immatriculation
Marque	VARCHAR(20)	Marque du véhicule
Modele	VARCHAR(20)	Modèle
Cylindree	SMALLINT	La cylindrée en cm <sup>3</sup>
Transmission	CHAR(1)	M pour boîte manuelle, A pour automatique
Prix	FLOAT	Coût d'une journée de location

Le répertoire client contiendra les colonnes suivantes :

Nom de colonne	Type	Commentaires
IdClient	SMALLINT	Numéro client unique
Nom	VARCHAR(40)	Le nom du client
Prenom	VARCHAR(40)	Son prénom
CodePostal	VARCHAR(10)	Code postal
Localite	VARCHAR(50)	Localité de résidence
Rue	VARCHAR(80)	Adresse
Numero	VARCHAR(10)	Numéro de maison
Telephone	VARCHAR(40)	Numéro de téléphone
Email	VARCHAR(50)	Adresse mail

Et voici les colonnes de la liste des locations :

Nom de colonne	Type	Commentaires
<b>IdLocation</b>	SMALLINT	Numéro d'ordre unique
<b>IdClient</b>	SMALLINT	Numéro du client
<b>Plaque</b>	VARCHAR(12)	Plaque de la voiture louée
DateDebut	DATETIME	Date et heure de début
DateFin	DATETIME	Date et heure de fin prévue
DateRentree	DATETIME	Date de rentrée effective du véhicule (pour surtaxe)
Assurance	BOOL	Indique si une assurance complémentaire a été prise

La plaque d'immatriculation et le numéro client étant uniques, ils sont désignés comme *clés primaires* dans leurs tables respectives et sont utilisés comme *clés étrangères* dans la table [Locations](#).

Collez la commande suivante pour créer la table [Voitures](#) :

```
CREATE TABLE Voitures (
  Plaque VARCHAR(12) NOT NULL,
  Marque VARCHAR(20) NOT NULL,
  Modele VARCHAR(20) NOT NULL,
  Cylindree SMALLINT NOT NULL,
  Transmission CHAR(1) NOT NULL,
  Prix FLOAT NOT NULL,
  PRIMARY KEY (Plaque)
);
```

Puis celle-ci pour créer le répertoire des clients (table [Clients](#)) :

```
CREATE TABLE Clients (
  IdClient SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  Nom VARCHAR(40) NOT NULL,
  Prenom VARCHAR(40) NOT NULL,
  CodePostal VARCHAR(10),
  Localite VARCHAR(50),
  Rue VARCHAR(80),
  Numero VARCHAR(10),
  Telephone VARCHAR(40) NOT NULL,
  Email VARCHAR(50),
  PRIMARY KEY (IdClient)
);
```

Et enfin pour la table [Locations](#) :

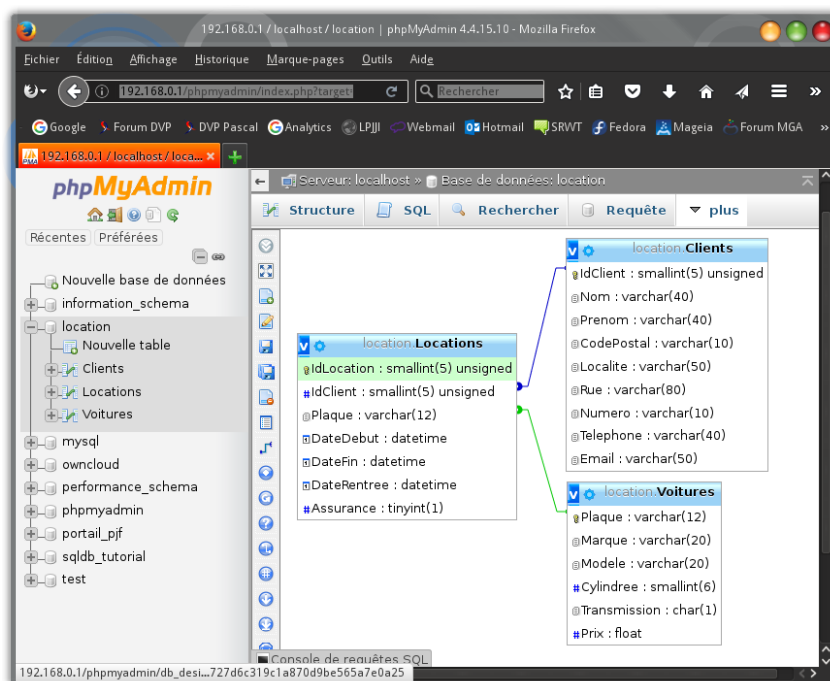
```
CREATE TABLE Locations (
  IdLocation SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  IdClient SMALLINT UNSIGNED NOT NULL,
  Plaque VARCHAR(12) NOT NULL,
  DateDebut DATETIME NOT NULL,
  DateFin DATETIME NOT NULL,
  DateRentree DATETIME,
  Assurance BOOL NOT NULL,
  PRIMARY KEY (IdLocation)
);
```

Établissons une relation entre la table [Locations](#) et les deux autres tables, par le biais des clés étrangères [IdClient](#) (table [Clients](#)) et [Plaque](#) (table [Voitures](#)) :

```
ALTER TABLE Locations ADD (
  CONSTRAINT FK_Locations_Voitures
  FOREIGN KEY (Plaque)
  REFERENCES Voitures(Plaque));
ALTER TABLE Locations ADD (
  CONSTRAINT FK_Locations_Clients
```

```
FOREIGN KEY (IdClient)
REFERENCES Clients(IdClient));
```

Dans phpMyAdmin, allez dans le sous-menu **Concepteur** du menu **Plus** de notre base de données **location**, pour vérifier que les relations sont correctes entre les trois tables :



## II-D - Création des données

Notre base de données est prête à être alimentée. Pour gagner du temps, nous allons y créer par script quelques voitures et quelques clients.

D'abord une dizaine de voitures :

```
INSERT INTO Voitures VALUES ('AB-612-BV', 'BMW', '135i Coupé', 3000, 'A', '30.00');
INSERT INTO Voitures VALUES ('AC-811-CK', 'Citroën', 'C3', 1600, 'M', '20.00');
INSERT INTO Voitures VALUES ('BN-101-AR', 'Citroën', 'C3', 1600, 'M', '20.00');
INSERT INTO Voitures VALUES ('AB-555-RB', 'Citroën', 'C3', 1600, 'M', '20.00');
INSERT INTO Voitures VALUES ('BU-425-GH', 'Citroën', 'C4 Coupe 2.0VTS', 2000, 'A', '27.50');
INSERT INTO Voitures VALUES ('AM-398-ER', 'Daihatsu', 'Sirion', 1300, 'M', '17.25');
INSERT INTO Voitures VALUES ('CB-135-RK', 'Fiat', '500 1.2 8V Lounge SS', 1200, 'M', '17.00');
INSERT INTO Voitures VALUES ('AM-400-SU', 'Fiat', 'Focus ST', 2000, 'M', '23.00');
INSERT INTO Voitures VALUES ('AM-436-FD', 'Ford', 'Taurus SHO', 3500, 'A', '36.25');
INSERT INTO Voitures VALUES ('CU-004-MP', 'Volkswagen', 'New Beetle', 2000, 'M', '27.50');
```

Puis quelques clients :

```
INSERT INTO Clients VALUES (1, 'Van Steenbrugge', 'Stefaan', 'B2020', 'Antwerpen (Belgique)', 'De Bosschaertstraat', '30', '3222012345', '');
INSERT INTO Clients VALUES (2, 'Leclercq', 'Jean-Jacques', '68200', 'Mulhouse', 'Avenue Aristide Briand', '37', '33389337878', 'jj.leclercq@monfai.fr');
INSERT INTO Clients VALUES (3, 'Lamoureux', 'Gabriel', '69008', 'Lyon', 'Rue Professeur Beauvisage', '163', '33472985404', 'gaby.lamoureux@monfai.fr');
INSERT INTO Clients VALUES (4, 'Loumrhari', 'Mohamed', '55270', 'Varennes-en-Argonne', 'Rue Louis XVI', '12', '33329807101', '');
INSERT INTO Clients VALUES (5, 'Mispelter', 'Yves', 'CH1002', 'Lausanne (Suisse)', 'Place de la Palud', '2', '41213152555', 'yves.mispelter@myprovider.ch');
INSERT INTO Clients VALUES (6, 'Zidane', 'Yasmina', '13014', 'Marseille', 'Rue Paul Cuxe', '72', '33491095656', 'yasmina.z@monfai.fr');
```



```
INSERT INTO Clients VALUES (7, 'Patulacci', 'Stéphane', '20304', 'Ajaccio', 'Avenue Antoine Serafini', '', '33495515253', '');
INSERT INTO Clients VALUES (8, 'Matombo Nguza Aniomba', 'Honorine', '13008', 'Marseille', 'Rue du Commandant Rolland', '125', '33491553778', '');
INSERT INTO Clients VALUES (9, 'Filucco', 'Martial', '59000', 'Lille', '', '', '33320495000', '');
```

Nous avons enfin un peu de matière pour commencer notre tutoriel.

### III - Exemple 1 : affichage simple du contenu d'une table

Dans Lazarus, créez un projet de type [Application](#). Dans l'inspecteur d'objets, renommez la fiche principale ([Form1](#) par défaut) en [MainForm](#).

#### III-A - TMySQLConnection

Dans la palette de composants, cherchez l'onglet [SQLdb](#), qui contient une série de composants permettant de se connecter à différents systèmes de bases de données. Parmi ceux-ci figurent une série de **connecteurs** pour différentes version de MySQL ; ils sont représentés par des petits dauphins.

Un **connecteur** va se charger d'assurer la connexion avec le système de bases de données. C'est à lui qu'il faudra fournir le **nom de l'hôte** qui héberge la base de données, le **nom de la base de données** ainsi que le **nom d'utilisateur** et le **mot de passe** permettant d'y accéder.

Cliquez sur le connecteur qui correspond à votre version de MySQL et déposez un composant sur votre fiche vierge. Ce composant ne sera pas visible à l'exécution du programme, donc vous pouvez le placer n'importe où sur la fiche. Comme votre version de MySQL n'est peut-être pas la même que celle que j'utilise au moment de la rédaction de ce tutoriel (5.6), renommez le composant en [MySQLConnection](#) en changeant sa propriété [Name](#) ; ainsi, toute référence de version aura disparu.

Pour ce premier projet, nous allons fortement simplifier les choses et définir les noms d'hôte, de base de données, d'utilisateur et le mot de passe directement dans les propriétés du connecteur.



*Dans une application sérieuse, ces propriétés seront définies au cours du programme.*

Définissez les propriétés suivantes dans l'inspecteur d'objets :

Propriété	Valeur
HostName	<a href="#">192.168.0.1</a> (dans ma configuration) ou <a href="#">localhost</a> (si votre serveur MySQL tourne sur votre machine)
DatabaseName	location
UserName	mysqldevp
Password	passmysqldevp

#### III-B - Quel connecteur pour MariaDB ?

Comme cela a été évoqué dans l'introduction, **MariaDB** est une déclinaison libre de MySQL de plus en plus présente sur le marché des SGBD.

Dans l'onglet [SQLdb](#) de la palette de composants, vous ne trouvez que des références aux versions de MySQL, alors quel connecteur choisir lorsque le serveur vous retourne la version 10.1 de MariaDB ? Jusqu'à la 5.5, les numéros de version coïncidaient, ensuite ils divergent :

MySQL	MariaDB
5.1 à 5.5	5.1 à 5.5
5.6	10.0
5.7	10.1

### III-C - TSQLTransaction

À présent, nous avons besoin d'un composant qui va s'occuper de la **transaction** avec la base de données.

Dans l'onglet [SQLdb](#), cliquez sur l'icône [TSQLTransaction](#) et déposez un composant à côté de votre connecteur.

Cliquez sur le premier composant [MySQLConnection](#) ; dans l'inspecteur d'objets, cherchez la propriété [Transaction](#) et choisissez le composant que vous venez d'ajouter dans la liste déroulante des transactions ([SQLTransaction1](#)).

Si vous inspectez les propriétés du composant [SQLTransaction](#), vous pouvez voir que Lazarus a automatiquement établi un lien avec le connecteur et avec la base de données.

### III-D - TSQLQuery

Maintenant, il faut ajouter sur la fiche, à côté des deux premiers composants, un autre composant invisible, [TSQLQuery](#) (toujours dans l'onglet [SQLdb](#)). Il va relayer les **commandes SQL** vers la base de données et héberger les données entrantes et sortantes dans un **dataset**.

Initialisez la propriété [Database](#) de ce nouveau composant en choisissant [MySQLConnection](#) dans la liste déroulante.

Nous disposons à présent de tous les outils nécessaires pour communiquer avec notre base de données.

### III-E - TDataSource

Nous allons afficher le contenu d'une table, par exemple celle des voitures. Pour ce faire, nous utiliserons un composant [TDBGrid](#), qui est un **tableau** spécialisé dans l'affichage de données issues d'une base de données.

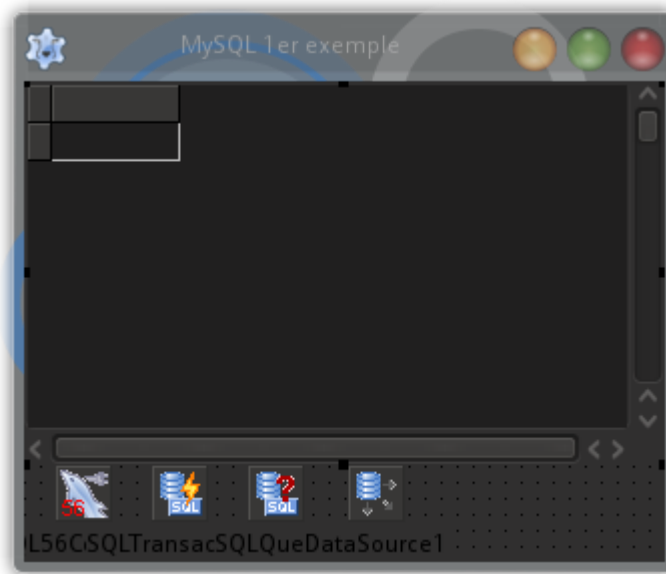
Ce tableau ne peut fonctionner seul ; il doit être couplé à un [TDataSource](#), qui va servir d'**intermédiaire entre le dataset et les contrôles** tels que le DBGrid.

Cherchez l'onglet [DataAccess](#), cliquez sur le [TDataSource](#) (normalement, le tout premier de la série) et déposez-en un exemplaire à côté des trois composants déjà installés (comme eux, il restera invisible).

Dans l'inspecteur d'objets, sa propriété [DataSet](#) doit être initialisée à [SQLQuery1](#) dans la liste déroulante.

### III-F - TDBGrid

Reste le tableau qui, lui, sera bien visible lors de l'exécution de l'application. Dans l'onglet [Data Controls](#), le [TDBGrid](#) est l'un des derniers. Déposez-le sur votre fiche et augmentez ses dimensions :



La propriété **DataSource** du DBGrid doit être initialisée à **DataSource1**, que nous avons déposé sur la fiche juste avant.

### III-G - Commande d'affichage d'une table

Bien, bien. Ce que nous voulons faire, c'est afficher dans notre tableau le contenu de la table **Voitures**. La commande SQL que nous devons exécuter sera :

```
SELECT * FROM Voitures;
```

Cette commande sera affectée à la propriété **SQL.Text** du composant **SQLQuery1**.

Pour commander l'affichage de la table, ajoutez un simple bouton (de l'onglet **Standard**). Dans l'inspecteur d'objets, initialisez sa propriété **Caption** à « Afficher ». Dans l'onglet **Événements** du composant, cliquez sur les trois points en regard de l'événement **OnClick** - cela créera une méthode événementielle dans le code source.



*Un double-clic sur la case vide à côté de l'événement a le même résultat.*

Voici le contenu de cette méthode :

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
    SQLQuery1.Close;
    SQLQuery1.SQL.Text:= 'SELECT * FROM Voitures;';
    SQLQuery1.Open;
end;
```

### III-H - Fermeture propre de l'application

Nous devons également prévoir la déconnexion propre de notre application à la base de données. Sélectionnez la fiche et, dans l'onglet **Événements** de l'inspecteur d'objets, cliquez sur les trois points en regard de **OnClose**.

Voici le code de la méthode événementielle créé :

```
procedure TMainForm.FormClose(Sender: TObject; var CloseAction: TCloseAction);
begin
    SQLQuery1.Close;
    SQLTransaction1.Active:= False;
    MySQLConnection.Connected:= False;
end;
```

Il est temps de sauvegarder le projet. Nommez-le (par exemple) `mysql01`, et enregistrez-le dans un répertoire du même nom.

### III-I - Compilation et exécution

Vérifions à présent que nous avons bien travaillé. Compilez et exécutez votre programme ; si vous n'avez pas commis d'erreur, le fait de presser sur le bouton [Afficher](#) remplit le DBGrid avec la liste des voitures :



Nous notons au passage que la gestion des caractères accentués (notre base de données a été créée avec le jeu de caractères **UTF8**) n'est pas correcte. Pour y remédier, dans l'inspecteur d'objets, il faut initialiser la propriété `CharSet` du connecteur `MySQLConnection` à `UTF8`.



*On s'attendrait a priori à trouver une liste déroulante avec tous les jeux de caractères disponibles, mais il n'en est rien : la valeur « UTF8 » doit être introduite au clavier.*

### III-J - Code complet de l'exemple 1

```
unit Main;

{$mode objfpc}{$H+}

interface

uses
    Classes, SysUtils, mysql56conn, sqldb, db, FileUtil, Forms, Controls,
    Graphics, Dialogs, DBGrids, StdCtrls;

type

    { TMainForm }
```

```

TMainForm = class(TForm)
    Button1: TButton;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    MySQLConnection: TMySQL56Connection;
    SQLQuery1: TSQLQuery;
    SQLTransaction1: TSQLTransaction;
    procedure Button1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var CloseAction: TCloseAction);
private
    { private declarations }
public
    { public declarations }
end;

var
    MainForm: TMainForm;


implementation

{$R *.lfm}

{ TMainForm }

procedure TMainForm.Button1Click(Sender: TObject);
(* Exécution de la requête d'affichage de la table *)
begin
    SQLQuery1.Close;
    SQLQuery1.SQL.Text:= 'SELECT * FROM Voitures;';
    SQLQuery1.Open;
end;
procedure TMainForm.FormClose(Sender: TObject; var CloseAction: TCloseAction);
(* Déconnexion propre de la base de données *)
begin
    SQLQuery1.Close;
    SQLTransaction1.Active:= False;
    MySQLConnection.Connected:= False;
end;
end.

```

Téléchargez le projet mysql01 complet  [ici](#).

## IV - Exemple 2 : édition d'une table au choix

Notre premier exemple nous a permis de nous familiariser avec les composants nécessaires à la communication avec une base de données MySQL.

Nous allons créer un second projet un peu plus élaboré, qui va nous permettre de sélectionner une des tables, de l'afficher dans un DBGrid et d'en modifier le contenu.

Créez un nouveau projet de type [Application](#).

Dans l'inspecteur d'objets, renommez la fiche principale (**Form1** par défaut) en **MainForm**. Agrandissez la taille de la fiche. Déposez-y, tout comme dans le premier exemple, les composants invisibles qui permettent de communiquer avec la base de données :

- un connecteur de la bonne version, que vous renommez directement en **MySQLConnection** ;
- un **TSQLTransaction** ;
- un **TSQLQuery**.

Cette fois, les propriétés **HostName**, **DatabaseName**, **UserName** et **Password** de **MySQLConnection** restent vierges de toute valeur initiale ; la propriété **Transaction** doit être initialisée à **SQLTransaction1** et la propriété **CharSet** doit être initialisée à **UTF8**. La propriété **Database** de **SQLQuery1** doit être initialisée à **MySQLConnection**.

## IV-A - Mot de passe de connexion

Nous allons faire en sorte que l'utilisateur du programme tape, au début du programme, le mot de passe permettant de se connecter à la base de données. Nous allons faire cela au moment où la fenêtre de l'application reçoit le focus.

Cliquez, dans l'inspecteur d'objets, sur la fiche elle-même, et allez dans l'onglet [Événements](#). Cliquez sur les trois points en regard de l'événement [OnActivate](#), ce qui aura pour effet de créer dans le code source la méthode événementielle [FormActivate](#).

Voici le contenu de cette méthode :

```
procedure TMainForm.FormActivate(Sender: TObject);
var
  LPassword : String;
begin
  MySQLConnection.HostName := '192.168.0.1';
  MySQLConnection.DatabaseName := 'location';
  MySQLConnection.UserName := 'mysqldvp';
  if InputQuery('Connexion à la base de données', 'Tapez votre mot de passe :', True, LPassword)
  then
    begin
      MySQLConnection.Password := LPassword;
      try
        MySQLConnection.Connected := True;
        SQLTransaction1.Active := True;
      except
        on e: EDatabaseError do
          begin
            MessageDlg('Erreur de connexion à la base de données.'#10#13'Le mot de passe est
            peut-être incorrect ?'#10#10#13'Fin de programme.', mtError, [mbOk], 0);
            Close;
          end;
        end;
      end;
    else
      (* Pas de mot de passe : fin de programme *)
      Close;
    end;
end;
```

Ajoutez également l'unité [db](#) à la clause [uses](#) (pour [EDatabaseError](#)).

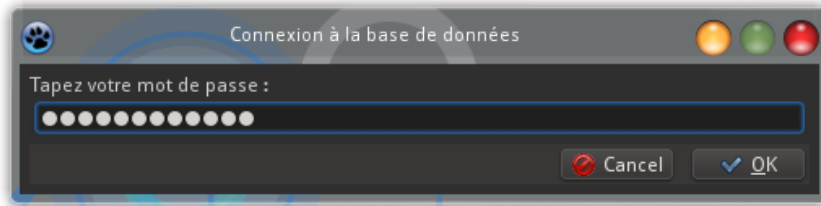
Détaillons tout cela. Pour commencer, nous initialisons les propriétés [HostName](#), [DatabaseName](#) et [UserName](#) du connecteur. Le mot de passe est demandé au moyen d'un dialogue [InputQuery](#). Si l'utilisateur annule l'entrée du mot de passe, le programme se ferme sans autre forme de procès ; s'il entre un mot de passe erroné, l'exception déclenchée par le connecteur est interceptée dans le bloc [try...except](#), un message d'erreur est affiché et l'application est fermée.

Avant de réaliser nos premiers tests à ce stade, n'oublions pas de faire en sorte que la connexion à la base de données soit proprement coupée à la fermeture du programme. Comme dans le premier exemple, créez une méthode événementielle pour l'événement [OnClose](#) :

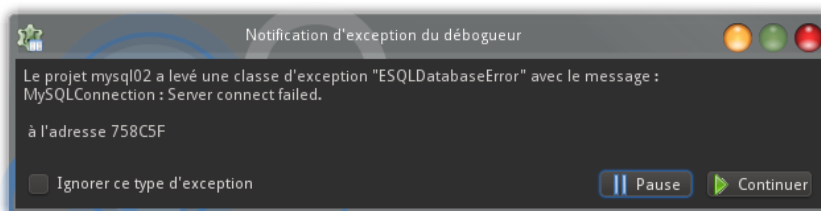
```
procedure TMainForm.FormClose(Sender: TObject; var CloseAction: TCloseAction);
begin
  SQLQuery1.Close;
  if SQLTransaction1.Active
  then
    SQLTransaction1.Active := False;
  if MySQLConnection.Connected
  then
    MySQLConnection.Connected := False;
end;
```

Enregistrez le projet, en l'appelant [mysql02](#) et en l'enregistrant dans un nouveau répertoire du même nom.

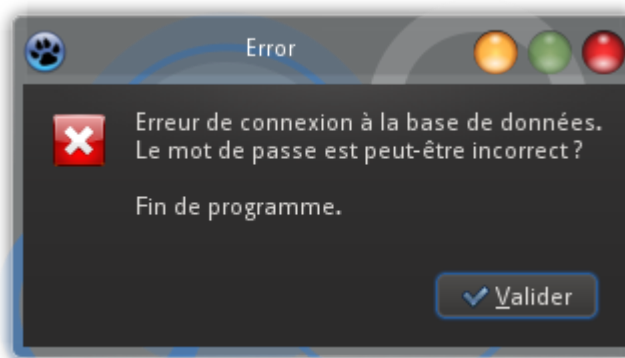
Compilez le programme et exécutez-le depuis l'EDI Lazarus. Vous constatez qu'une fenêtre vierge apparaît et qu'immédiatement le dialogue [InputQuery](#) vous demande le mot de passe :



Si vous tapez le mot de passe correct (« passmysqldvp », rappelez-vous), la fenêtre de l'application (qui ne contient rien de visible pour l'instant) reste ouverte ; si vous cliquez sur [Cancel](#), l'application se ferme et si vous tapez exprès un mot de passe erroné, un message d'erreur apparaît :



Tiens, mais ce n'est pas le message d'erreur que nous avons prévu dans la méthode [FormActivate](#) ? En effet, mais si vous cliquez sur le bouton [Continuer](#), celui que nous avons prévu apparaît bien :



Pourquoi ? Tout simplement parce que le premier message d'erreur est renvoyé par le débogueur, qui lève le premier l'exception.

Plutôt que d'exécuter le programme depuis l'EDI, utilisez l'exécutable qui a été créé dans le répertoire [mysql02](#).

Exécutez-le et faites l'expérience d'entrer un mot de passe erroné : c'est bien le message d'erreur que nous avons prévu qui s'affiche, puisque l'exécution du programme n'est plus encadrée par le débogueur.

#### IV-A-1 - Affichage de l'erreur renvoyée par le système

Notre message d'erreur n'est pas très précis, car il apparaît identiquement si le mot de passe est erroné et si, par exemple, le serveur est inaccessible ou la base de données inexistante.

Pour pouvoir identifier plus précisément l'erreur qui s'est produite, il faut lever une exception descendante de [EDatabaseError](#) : [ESQLDatabaseError](#). Dans les propriétés de ce type d'exception se trouvent un code et des messages propres à MySQL.

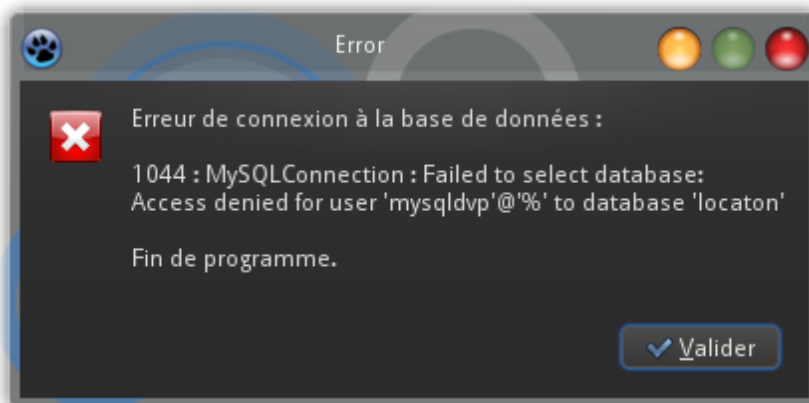


Une liste des erreurs possibles est consultable dans la [référence de MySQL en ligne](#).

Voici une version plus élaborée de notre méthode de login :

```
procedure TMainForm.FormActivate(Sender: TObject);
(* Lecture du mot de passe et connexion à la base de données *)
var
  LPassword : String;
begin
  (* Données de la connexion *)
  MySQLConnection.HostName := '192.168.0.1';
  MySQLConnection.DatabaseName := 'location';
  MySQLConnection.UserName := 'mysqldvp';
  (* Lecture du mot de passe *)
  if InputQuery('Connexion à la base de données', 'Tapez votre mot de passe :', True, LPassword)
  then
    begin
      (* Connexion à la base de données *)
      MySQLConnection.Password := LPassword;
      try
        MySQLConnection.Connected := True;
        SQLTransaction1.Active := True;
      except
        on e: ESQLDatabaseError do
          begin (* Erreur renvoyée par MySQL : fin de programme *)
            MessageDlg('Erreur de connexion à la base de données :'#10#10#13 +
              IntToStr(e.ErrorCode) + ' : ' + e.Message +
              '#10#10#13'Fin de programme.', mtError, [mbOk], 0);
            Close;
          end;
        on e: EDatabaseError do
          begin (* Erreur de connexion : fin de programme *)
            MessageDlg('Erreur de connexion à la base de données.'#10#10#13'Fin de
              programme.', mtError, [mbOk], 0);
            Close;
          end;
        end;
      end
    else (* Pas de mot de passe : fin de programme *)
      Close;
    end;
end;
```

Par exemple, si la base de données n'existe pas :





La gestion de l'exception [EDatabaseError](#) suit celle de [ESQLDatabaseError](#). Le principe général est de gérer en cascade les exceptions de la plus spécialisée à la moins spécialisée : ainsi, si la connexion au serveur MySQL est possible, les erreurs MySQL seront traitées en priorité ([ESQLDatabaseError](#)).



***Vous avez certainement remarqué que les messages d'erreur sont en anglais, la langue par défaut de Lazarus. Pour obtenir les messages en français, je vous renvoie au tutoriel de Gilles Vasseur sur l'[internationalisation d'une application](#).***

## IV-B - Récupération de la liste des tables

À présent, nous allons faire en sorte que l'utilisateur puisse choisir une des tables contenues dans la base de données pour en afficher le contenu.

Sélectionnez un composant [TListBox](#) dans l'onglet [Standard](#) et déposez-le en bas et à gauche sur la fiche du projet. Dans l'inspecteur d'objets, renommez sa propriété [Name](#) en [lbTables](#).

Dans l'éditeur de source, allez dans la déclaration du type [TMainForm](#) (que Lazarus a automatiquement déclaré ainsi lorsque vous avez appelé [MainForm](#) votre fiche principale). Dans la section [private](#), ajoutez cette procédure :

```
procedure ShowTables;
```

Pressez la combinaison de touches **Shift-Control-C** pour que Lazarus crée la procédure dans la section [implementation](#) et complétez cette dernière :

```
procedure TMainForm.ShowTables;
begin
  SQLQuery1.Close;
  SQLQuery1.SQL.Text := 'SHOW TABLES;';
  SQLQuery1.Open;
  while not SQLQuery1.EOF do
  begin
    lbTables.Items.Add(SQLQuery1.Fields[0].AsString);
    SQLQuery1.Next;
  end;
  (* Sélection du premier élément *)
  if SQLQuery1.RecordCount > 0
  then
    lbTables.ItemIndex := 0;
end;
```

Tout d'abord, la commande SQL qui permet de retourner la liste des tables d'une base de données est [SHOW TABLES](#);

Cette commande est passée au composant [SQLQuery1](#) dans sa propriété [SQL.Text](#) (comme nous avons fait dans notre premier exemple) et est exécutée dans la méthode [Open](#). Ensuite, nous bouclons pour que chaque nom de table retourné par la requête soit ajouté dans la *listbox* [lbTables](#).

Placez l'appel de cette méthode privée [ShowTables](#) dans le bloc [try](#) de la méthode événementielle [FormActivate](#) de la fiche principale :

```
procedure TMainForm.FormActivate(Sender: TObject);
var
  LPassword : String;
begin
  { . . . }
  if InputQuery('Connexion à la base de données', 'Tapez votre mot de passe :', True, LPassword)
  then
    begin
```

```
MySQLConnection.Password := LPassword;
try
  MySQLConnection.Connected := True;
  SQLTransaction1.Active := True;
  ShowTables; // ***** AJOUT *****
except
  { . . . }
end;
end
else (* Pas de mot de passe : fin de programme *)
  Close;
end;
```

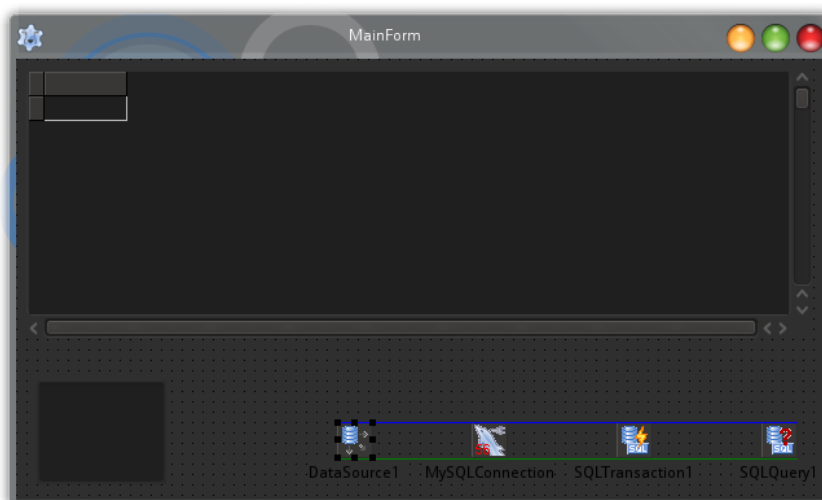
Vous pouvez faire un test d'exécution à ce stade, pour vérifier que la liste des tables est bien chargée dans la *listbox* :



#### IV-C - Affichage du contenu de la table sélectionnée

Comme dans le premier exemple, nous allons afficher le contenu d'une table dans un composant **TDBGrid**, de l'onglet **Data Controls**, que vous déposez en haut et à gauche de votre fiche principale, et dont vous agrandissez la largeur jusqu'à la limite droite de la fiche et la hauteur jusqu'à un peu au-dessus de la *listbox*.

L'utilisation du DBGrid nous conduit à ajouter également un composant **TDataSource**, de l'onglet **Data Access** :



Toujours comme dans l'exemple 1, la propriété `DataSet` du `TDataSource` doit être affectée à `SQLQuery1`, et la propriété `DataSource` du `TDBGrid` à `DataSource1`.

Dans l'inspecteur d'objets, sélectionnez la `listbox lbTables`, allez dans l'onglet `Événements`, trouvez l'événement `OnSelectionChange` et cliquez sur les trois points en regard de celui-ci pour créer une méthode événementielle qui s'exécutera à chaque changement de choix de table :

```
procedure TMainForm.lbTablesSelectionChange(Sender: TObject; User: boolean);
begin
    SQLQuery1.Close;
    SQLQuery1.SQL.Text := 'SELECT * FROM ' + lbTables.GetSelectedText + ';';
    SQLQuery1.Open;
end;
```

Simplement, la requête `SELECT * FROM` est complétée par le nom de la table sélectionnée dans la `listbox`.

Testez l'application en l'état :



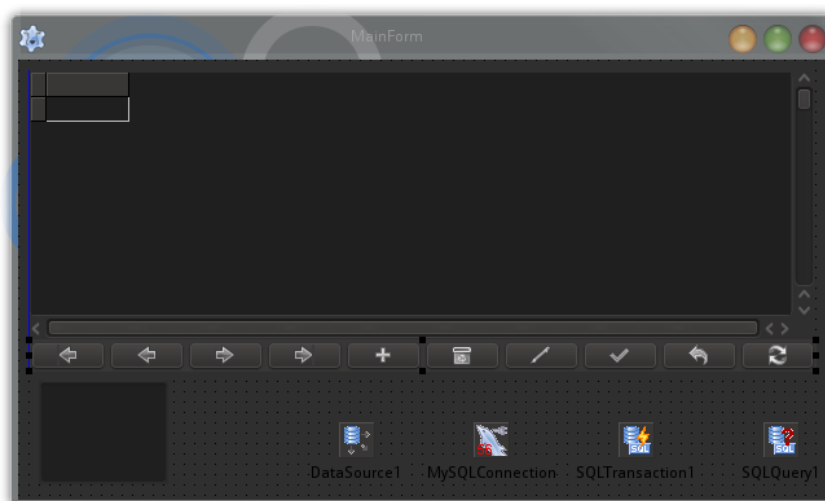
Le fait de sélectionner une autre table charge automatiquement son contenu dans le `DBGrid`.



*Si vous êtes très attentif(ve), vous verrez très fugacement apparaître la liste des tables dans le `DBGrid`, juste avant que le contenu de la première table s'affiche. C'est dû au fait que le `DBGrid`, via le `DataSource`, affiche le contenu du dataset du `SQLQuery`. Pour éviter cette brève apparition, il aurait fallu ne pas lier le `DBGrid` au `DataSource` dans ses propriétés et ajouter cette instruction juste après l'exécution de `ShowTables` (dans la méthode `FormActivate`) : `DBGrid1.DataSource := DataSource1;`*

## IV-D - Permettre de modifier les données

Nous avons laissé un peu de place entre le bord supérieur de la `listbox` et le bord inférieur du `DBGrid` pour pouvoir y insérer une barre d'outils de navigation : un composant `TDBNavigator` (que l'on trouve au début de l'onglet `Data Controls`). Centrez-le ou donnez-lui la même largeur que le `DBGrid` :



Affectez [DataSource1](#) à sa propriété [DataSource](#). Dans l'inspecteur d'objets, vous pouvez choisir quels boutons vous voulez afficher dans le DBNavigator, dans sa propriété [VisibleButtons](#). Décochez notamment le bouton [nbRefresh](#), qui n'a aucune utilité dans notre exemple.

Exécutez à nouveau l'application et expérimentez la navigation avec les boutons verts, mais aussi l'insertion de lignes, la suppression, etc. Vous pouvez y aller franchement et tout casser : les modifications que vous faites affectent juste le contenu du DBGrid, mais pas la base de données elle-même. D'ailleurs, si vous changez votre choix de table dans la *listbox*, vous voyez qu'à chaque réaffichage la table est restaurée dans son état d'origine.

La dernière étape va être d'enregistrer les modifications dans la base de données.



*Ne martyrisez pas trop la base de données dans les tests que vous ferez dorénavant, car nous en aurons encore besoin pour la suite du tutorial.*

Nous allons donc écrire une méthode privée qui va s'occuper de mettre à jour les données des tables dans la base. Cette méthode sera appelée à chaque fois que l'on changera de table dans la *listbox*, ainsi qu'à la fermeture de l'application.

Retournez dans l'éditeur de source et ajoutez cette méthode dans la section [private](#) de [TMainForm](#) :

```
procedure CommitChanges;
```

Un petit **Shift-Control-C** pour son implémentation :

```
procedure TmainForm.CommitChanges;
begin
    if SQLTransaction1.Active
    then
        try
            SQLQuery1.ApplyUpdates;
            SQLTransaction1.Commit;
        except
            on e: EDatabaseError do
                begin
                    MessageDlg('Erreur d'enregistrement des modifications', mtError, [mbOk], 0);
                end;
            end;
        end;
end;
```

Appelons-la dans la méthode événementielle qui répond au changement de table dans la *listbox* :

```
procedure TMainForm.lbTablesSelectionChange(Sender: TObject; User: boolean);
begin
    CommitChanges; // ***** AJOUT *****
    SQLQuery1.Close;
    SQLQuery1.SQL.Text := 'SELECT * FROM ' + lbTables.GetSelectedText + ';';
    SQLQuery1.Open;
end;
```

Et dans la méthode qui répond à la fermeture de la fiche principale :

```
procedure TMainForm.FormClose(Sender: TObject; var CloseAction: TCloseAction);
begin
    CommitChanges; // ***** AJOUT *****
    SQLQuery1.Close;
    if SQLTransaction1.Active
    then
        SQLTransaction1.Active := False;
    if MySQLConnection.Connected
    then
        MySQLConnection.Connected := False;
end;
```

Testez l'application ainsi modifiée (encore une fois, en y allant *mollo* pour garder utilisable la base de données). Inspectez le contenu de la base avec phpMyAdmin : vous constaterez que toutes les modifications y ont bien été répercutées.

## IV-E - Code complet de l'exemple 2

```
unit Main;

{$mode objfpc}{$H+}

interface

uses
    Classes, SysUtils, mysql56conn, sqldb, FileUtil, Forms, Controls, Graphics,
    Dialogs, StdCtrls, Grids, DBGrids, DbCtrls, db;

type

    { TMainForm }

    TMainForm = class(TForm)
        DataSource1: TDataSource;
        DBGrid1: TDBGrid;
        DBNavigator1: TDBNavigator;
        lbTables: TListBox;
        MySQLConnection: TMySQL56Connection;
        SQLQuery1: TSQLQuery;
        SQLTransaction1: TSQLTransaction;
        procedure FormActivate(Sender: TObject);
        procedure FormClose(Sender: TObject; var CloseAction: TCloseAction);
        procedure lbTablesSelectionChange(Sender: TObject; User: boolean);
    private
        { private declarations }
        procedure ShowTables;
        procedure CommitChanges;
    public
        { public declarations }
    end;

var
    MainForm: TMainForm;

implementation

{$R *.lfm}
```

```
{ TMainForm }

procedure TMainForm.FormClose(Sender: TObject; var CloseAction: TCloseAction);
(* Fermeture propre de la connexion avant la fin de programme *)
begin
    (* Enregistrement des éventuelles modifications *)
    CommitChanges;
    (* Fermeture de la connexion *)
    SQLQuery1.Close;
    if SQLTransaction1.Active
    then
        SQLTransaction1.Active := False;
    if MySQLConnection.Connected
    then
        MySQLConnection.Connected := False;
end;

procedure TMainForm.lbTablesSelectionChange(Sender: TObject; User: boolean);
(* Sélection d'une table dans la listbox *)
begin
    (* Enregistrement des éventuelles modifications *)
    CommitChanges;
    (* Chargement des données de la table choisie *)
    SQLQuery1.Close;
    SQLQuery1.SQL.Text := 'SELECT * FROM ' + lbTables.GetSelectedText + ';';
    SQLQuery1.Open;
end;

procedure TMainForm.ShowTables;
(* Chargement de la liste des tables dans la listbox *)
begin
    SQLQuery1.Close;
    SQLQuery1.SQL.Text := 'SHOW TABLES;';
    SQLQuery1.Open;
    while not SQLQuery1.EOF do
    begin
        lbTables.Items.Add(SQLQuery1.Fields[0].AsString);
        SQLQuery1.Next;
    end;
    if SQLQuery1.RecordCount > 0
    then
        lbTables.ItemIndex := 0;
end;

procedure TMainForm.CommitChanges;
(* Enregistrement des modifications *)
begin
    if SQLTransaction1.Active
    then
        try
            SQLQuery1.ApplyUpdates;
            SQLTransaction1.Commit;
        except
            on e: EDatabaseError do
            begin
                MessageDlg('Erreur d'enregistrement des modifications', mtError, [mbOk], 0);
            end;
        end;
end;

procedure TMainForm.FormActivate(Sender: TObject);
(* Lecture du mot de passe et connexion à la base de données *)
var
    LPassword : String;
begin
    (* Données de la connexion *)
    MySQLConnection.HostName := '192.168.0.1';
    MySQLConnection.DatabaseName := 'location';
    MySQLConnection.UserName := 'mysqldvp';
    (* Lecture du mot de passe *)
    if InputQuery('Connexion à la base de données', 'Tapez votre mot de passe : ', True, LPassword)
    then
        begin
```

```

(* Connexion à la base de données *)
MySQLConnection.Password := LPassword;
try
    MySQLConnection.Connected := True;
    SQLTransaction1.Active := True;
    ShowTables;
except
    on e: EDatabaseError do
        begin (* Erreur de connexion : fin de programme *)
            MessageDlg('Erreur de connexion à la base de données.'#10#13'Le mot de passe est
peut-être incorrect ?'#10#13'Fin de programme.', mtError, [mbOk], 0);
            Close;
        end;
    end;
else (* Pas de mot de passe : fin de programme *)
    Close;
end;
end.

```

Téléchargez le projet mysql02  [ici](#).

## V - Exemple 3 : une application complète

Nous pourrions déjà arrêter là ce tutoriel : vous avez découvert les composants natifs de Lazarus permettant de créer des applications utilisant une base de données et vous pourrez sans trop de difficultés appliquer les principes vus pour MySQL à d'autres systèmes de bases de données. Mais nous allons essayer d'aller un peu plus loin, en découvrant d'autres contrôles spécialisés, en voyant comment manipuler les données dans des contrôles classiques (non spécialisés dans les bases de données), et comment regrouper le traitement des données dans une unité d'un type un peu particulier : un **DataModule**. Nous parlerons également de l'intérêt de centraliser la génération des requêtes SQL dans une **interface**.

### V-A - Création du projet

Commençons par le commencement :

- créez un nouveau projet de type **Application** ;
- renommez l'unité **Unit1** en **Main** ;
- renommez la fiche principale en **MainForm** (qui devient automatiquement de type **TMainForm**) ;
- changez sa propriété **Caption** (son titre) en, par exemple, « Location de voitures » ;
- enregistrez le projet sous le nom **mysql03**, dans un nouveau répertoire du même nom.

### V-B - Unité de type DataModule

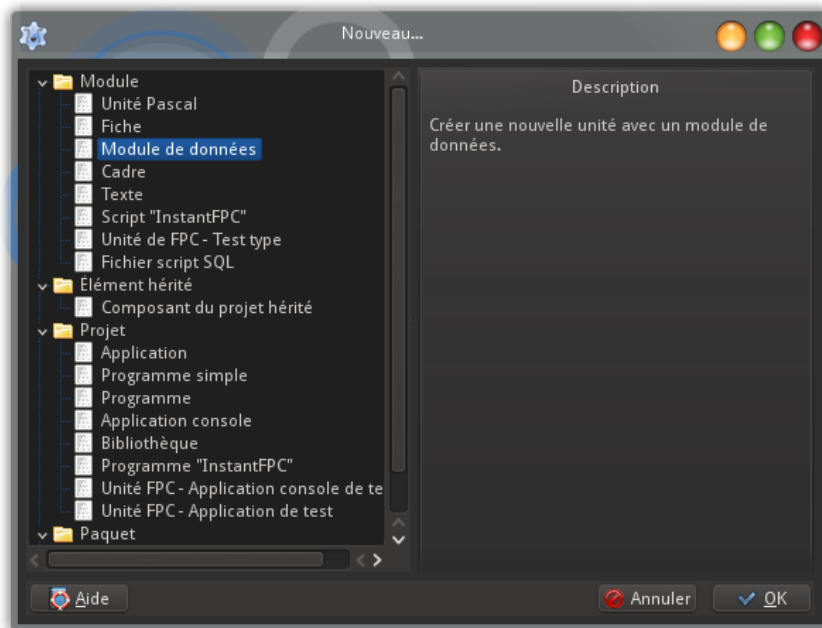
Les deux premiers exemples étaient de minuscules applications, qui ne sont pas très difficiles à comprendre ni à déboguer, pour un développeur qui les découvre. Lorsque l'on crée des applications de plus grande ampleur, ces aspects (comprendre et déboguer) prennent toute leur importance et il faut que vous aussi vous y retrouviez facilement si vous devez en assurer la maintenance dans le futur.

Nous allons faire en sorte que tout le traitement en rapport avec la base de données soit regroupé à part, et Lazarus possède un type particulier d'unité adapté à cela : le **DataModule**. Ce type d'unité est l'endroit idéal pour déposer des composants invisibles, comme ceux que nous avons utilisés dans ce tutoriel, mais ce n'est pas limitatif.



*Cette manière de faire pourrait aussi faciliter la migration d'une application vers un autre système de gestion de bases de données.*

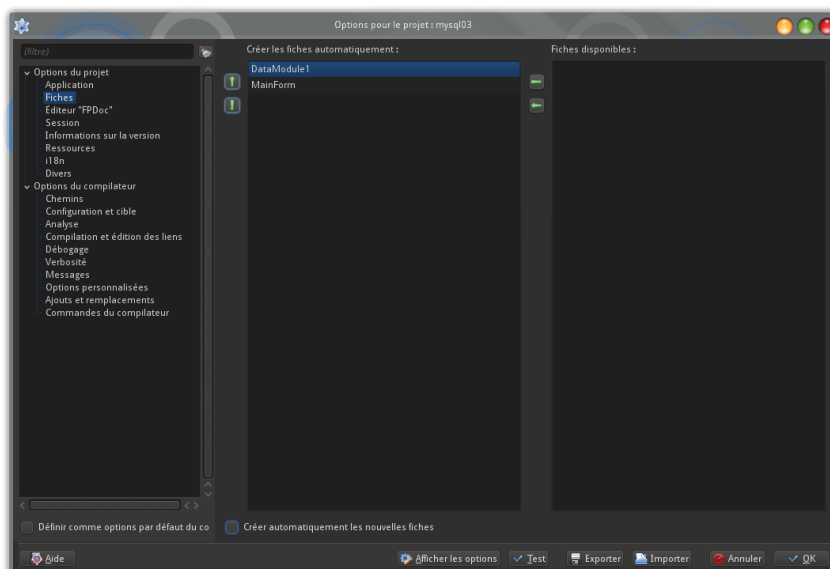
Allez dans le menu **Fichier, Nouveau, Module de données** et cliquez sur **OK** pour ajouter une unité DataModule :



La nouvelle unité créée présente une fiche similaire à une fiche normale de type **Tform**. Renommez tout de suite l'unité en **DataAccess**.

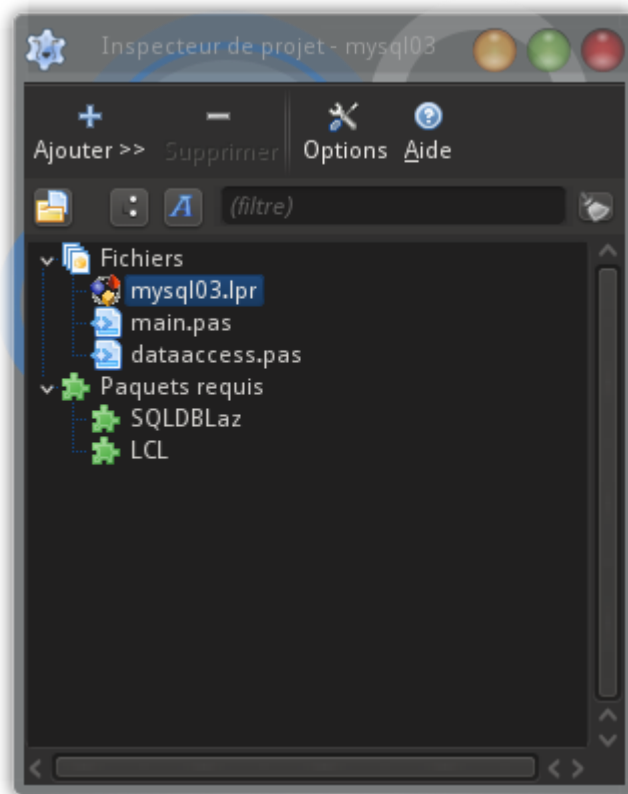
Depuis l'onglet de composants **SQLdb**, déposez sur la fiche un composant **TMySQLxxConnection** correspondant à votre version, et un composant **TSQLTransaction**. Renommez-les respectivement **SQLConnection** et **SQLTransaction**, et assignez **SQLConnection** à la propriété **Database** de **SQLTransaction**. N'oubliez pas d'inscrire **UTF8** dans la propriété **CharSet** de **SQLConnection**.

Allez dans le menu **Projet, Options du projet** et cliquez sur l'entrée **Fiches**. Vous constatez que la fiche principale et le *datamodule* sont automatiquement créés au démarrage de l'application. C'est très bien ainsi, hormis qu'ils ne sont pas créés dans le bon ordre : vous comprendrez très vite pourquoi le *datamodule* doit être créé avant la fiche principale. Mettez **DataModule1** en surbrillance et faites-le monter en tête de liste à l'aide de la petite flèche verte à gauche. Tant que vous y êtes, décochez la case **Créer automatiquement les nouvelles fiches** au bas du dialogue : nous n'aurons pas besoin que toutes les autres fiches de l'application soient créées au démarrage.





Cliquez sur **OK**. Ouvrez l'inspecteur de projet, par le biais du menu **Projet, Inspecteur de projet** :



Le projet est pour l'instant composé d'un programme principal, **mysql03.lpr**, et de deux unités, **main.pas** et **dataaccess.pas**. Double-cliquez sur le programme principal : dans son code source, vous voyez que le *datamodule* est bien créé avant la fiche principale :

```
program mysql03;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Interfaces, // this includes the LCL widgetset
  Forms,
  DataAccess,
  Main
  { you can add units after this };
{$R *.res}
begin
  RequireDerivedFormResource:=True;
  Application.Initialize;
  Application.CreateForm(TDataModule1, DataModule1);
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

Nous avons mentionné, au début de ce chapitre, que l'utilisation du *datamodule* pourrait faciliter la migration de notre application vers un autre système de gestion de bases de données, comme SQLite ou PostgreSQL. Alors nous allons jouer le jeu et y regrouper **tout ce qui est propre à MySQL**.

Nous allons d'abord y inclure les méthodes de connexion (avec la demande de mot de passe) et de déconnexion que nous avons développées dans l'exemple 2.

Cliquez sur l'onglet du code source de l'unité [DataAccess](#) et ajoutez, dans la section **public** de la classe [TDataModule1](#), ces deux méthodes :

```
function Login : Boolean;
procedure Logoff;
```

Pressez la combinaison de touches **Shift-Ctrl-C**, afin que Lazarus crée les deux méthodes dans la section **implementation**.

Recopiez dans la méthode [Login](#) le code de connexion contenu dans la méthode [FormActivate](#) de l'exemple 2, et dans la méthode [Logoff](#) le contenu de la méthode [FormClose](#) :

```
function TDataModule1.Login: Boolean;
(* Demande du mot de passe et connexion à la base de données *)
var
  LPassword : String;
begin
  Result := True;
  SQLConnection.HostName := '192.168.0.1';
  SQLConnection.DatabaseName := 'location';
  SQLConnection.UserName := 'mysqldvp';
  if InputQuery('Connexion à la base de données', 'Tapez votre mot de passe :', True, LPassword)
  then
    begin
      SQLConnection.Password := LPassword;
      try
        SQLConnection.Connected := True;
        SQLTransaction.Active := True;
      except
        on e : ESQLDatabaseError do
          begin (* Erreur renvoyée par MySQL : fin de programme *)
            MessageDlg('Erreur de connexion à la base de données :'#10#10#13 +
              IntToStr(e.ErrorCode) + ' : ' + e.Message +
              #10#10#13'Fin de programme.', mtError, [mbOk], 0);
            Result := False;;
          end;
        on e : EDatabaseError do
          begin (* Erreur de connexion : fin de programme *)
            MessageDlg('Erreur de connexion à la base de données.'#10#10#13'Fin de
programme.', mtError, [mbOk], 0);
            Result := False;;
          end;
        end;
      else
        Result := False;
      end;
    end;
end;
procedure TDataModule1.Logoff;
(* Déconnexion de la base de données *)
begin
  if SQLTransaction.Active
  then
    SQLTransaction.Active := False;
  if SQLConnection.Connected
  then
    SQLConnection.Connected := False;
end;
```

Pour que le compilateur trouve la fonction [InputQuery](#), ajoutez l'unité [Dialogs](#) à la clause **uses** du *datamodule*, et ajoutez l'unité [db](#) pour le type [EDatabaseError](#).

Retournez dans l'unité [Main](#) et pressez **F12** pour afficher la fiche principale. Dans l'inspecteur d'objets, dans l'onglet [Événements](#), descendez sur l'événement [OnShow](#) et cliquez sur les trois points correspondants. Cela va créer la méthode [TMainForm.FormShow](#) dans la section implémentation. C'est à cet endroit que nous allons appeler la méthode de login que nous avons implémentée dans le *datamodule* :

```
procedure TMainForm.FormShow(Sender: TObject);
(* Demande de mot de passe *)
begin
if DataModule1.Login
then
begin
ShowMessage('Login couronné de succès !');
end
else
Close;
end;
```

Faites la même chose avec l'événement **OnClose** :

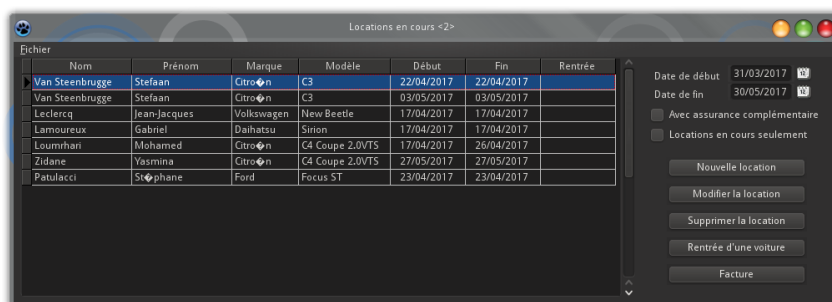
```
procedure TMainForm.FormClose(Sender: TObject; var CloseAction: TCloseAction);
(* Fermeture propre de la connexion à la base de données *)
begin
DataModule1.Logoff;
end;
```

Il ne faut pas oublier d'ajouter l'unité **DataAccess** à la clause **uses** de l'unité **Main**, sinon le compilateur dira qu'il ne connaît ni **Login** ni **Logoff**.

Vous pouvez compiler et exécuter votre application à ce stade, pour vérifier que la connexion est bien couronnée de succès.

Bon, il est temps de définir à quoi va ressembler et ce que va faire notre application de gestion de location de voitures.

Nous aurons une fenêtre principale, qui va contenir la liste des locations :



Un dialogue permettra de gérer la liste des voitures disponibles :

Liste des voitures <2>

Plaque	Marque	Modèle	Cyl.	Trans.	€/jour
AB-555-RB	Citroën	C3	1600	M	20
AB-612-BV	BMW	135i Coupé	3000	A	30
AC-811-CK	Citroën	C3	1600	M	20
AM-398-ER	Daihatsu	Sirion	1300	M	17.25
AM-400-SU	Ford	Focus ST	2000	M	23
AM-436-FD	Ford	Taurus SHO	3500	A	36.25
BN-101-AR	Citroën	C3	1600	M	20
BU-425-GH	Citroën	C4 Coupe 2.0VTS	2000	A	27.5
CB-135-RK	Fiat	500 1.2 8V Lounge	1200	M	17
CU-004-MP	Volkswagen	New Beetle	2000	M	27.5

Plaque: AB-555-RB  
 Marque: Citroën  
 Modèle: C3  
 Cylindrée: 1600  
 €/jour: 20

Transmission:  
☐ A  
☒ M

Annuler Enregistrer

Un autre, similaire, sera consacré à la liste des clients :

Liste des clients <2>

Nom	Prénom
Van Steenbrugge	Stefaan
Leclercq	Jean-Jacques
Lamoureux	Gabriel
Loumrhari	Mohamed
Mispelter	Yves
Zidane	Yasmina
Patulacci	Stéphane
Matombo Nguza Aniomba	Honorine
Filucco	Martial

Nom: Van Steenbrugge  
 Prénom: Stefaan  
 Adresse: De Bosschaertstraat 30  
 B2020 Antwerpen (Belgique)  
 Téléphone: 3222012345  
 E-mail:

Annuler Enregistrer

Dans un dialogue, on pourra créer une nouvelle location :

Ce dialogue servira également à modifier une location existante.

Pour terminer, nous sortirons une facture à l'aide d'un générateur de rapports.

Nous avons du pain sur la planche ! Créons tout de suite le dialogue de gestion des voitures.

## V-C - Utilisation des composants spécialisés

Dans les deux premiers exemples de ce tutoriel, vous avez déjà découvert les composants [TDBGrid](#) et [TDBNavigator](#). Nous allons encore nous en servir, mais nous allons aussi utiliser d'autres composants spécialisés comme [TDBEdit](#) et [TDBRadioGroup](#). Si vous parcourez l'onglet [Data Controls](#) de la palette, vous trouvez toute une panoplie de composants : mémo, liste déroulante, etc. Avec ceux que nous allons voir ici, vous devriez être à même de les utiliser tous par la suite.

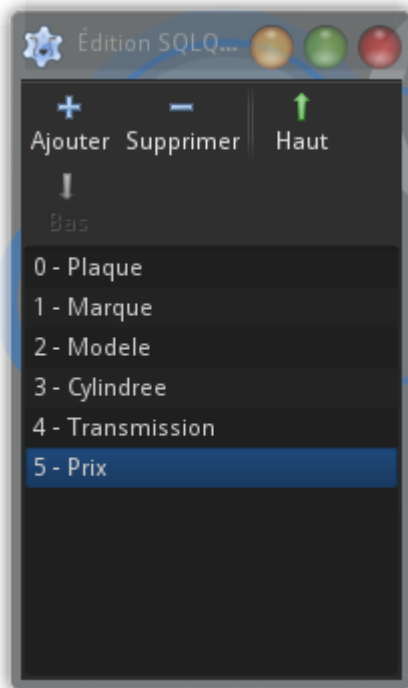
L'utilisation de tous ces composants spécialisés impose de mettre en service, comme dans les deux premiers exemples, un [TSQLQuery](#) et un [TDataSource](#). Et à quel endroit allons-nous les placer ? Dans le *datamodule*, bien sûr !



*Nous assignerons un [TSQLQuery](#) et, éventuellement, un [TDataSource](#) à chaque table de la base de données.*

Cliquez sur l'onglet du code source de l'unité [DataAccess](#) puis pressez **F12**. Sur la fiche [DataModule1](#), déposez donc un exemplaire de chacun de ces deux composants (le [TSQLQuery](#) depuis l'onglet [SQLdb](#) et le [TDataSource](#) depuis l'onglet [Data Access](#)). Renommez-les respectivement [SQLQueryVoitures](#) et [DataSourceVoitures](#). La propriété [Database](#) du premier doit être initialisée à [SQLConnection](#) et la propriété [DataSet](#) du second à [SQLQueryVoitures](#).

Tant que nous sommes dans l'inspecteur d'objets, nous allons définir les différents champs de la table [Voitures](#) dans les propriétés de [SQLQueryVoitures](#). Repérez la propriété [FieldDefs](#) et cliquez sur les trois points en regard. Un dialogue va s'ouvrir, dans lequel vous allez ajouter successivement les champs en cliquant sur le bouton « + » :



Pour chaque champ, vous allez définir dans l'inspecteur d'objets les propriétés **Name**, **DataType** et, éventuellement, **Size**. Voici la liste de ces propriétés :

Name	DataType	Size
Plaque	ftString	12
Marque	ftString	20
Modele	ftString	20
Cylindree	ftInteger	0
Transmission	ftFixedChar	1
Prix	ftFloat	0

Créons à présent une nouvelle fiche, à l'aide du second bouton de la barre d'outils. Concomitamment, une nouvelle unité est créée : renommez-la **Voitures** et enregistrez-la sous le nom **voitures.pas**. Dans l'explorateur d'objets, changez la propriété **Name** de la fiche en **CarForm** et indiquez son titre (par exemple, « Liste des voitures ») dans la propriété **Caption**.

Cliquez sur l'onglet de composants **Data Controls** de la palette. Depuis cet onglet, déposez sur la fiche les composants énumérés ci-après.

*Ne faites pas trop vite le lien entre les composants que vous allez déposer et le **TDataSource** : si vous le faites, Lazarus vous bloquera lorsque vous voudrez assigner aux contrôles les champs de la table **Voitures**.*

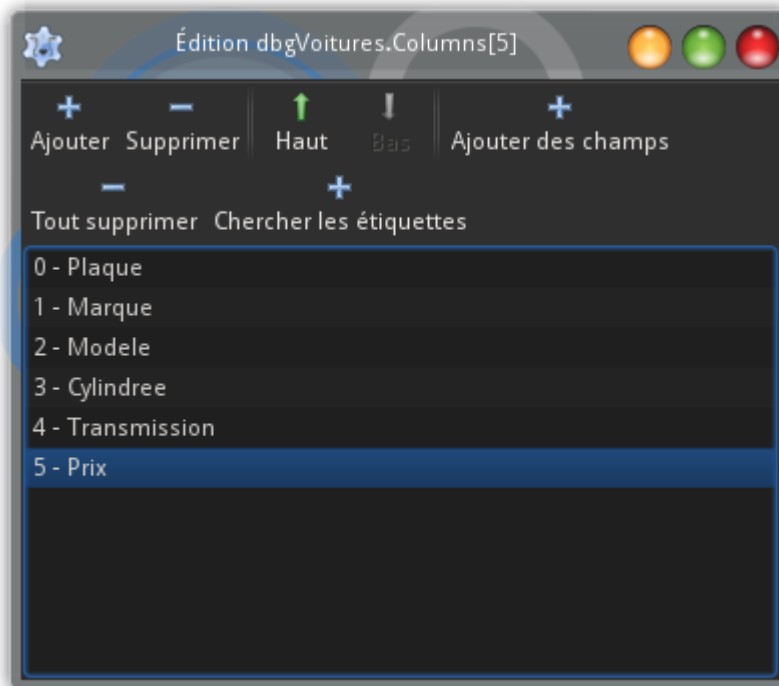


*Donc définissez d'abord toutes les autres propriétés de vos contrôles et finissez par leur propriété **DataSource**.*

## V-C-1 - TDBGrid

Nommez-le **dbgVoitures** dans sa propriété **Name** et dimensionnez-le à 424 pixels de largeur (**Width**) sur 240 pixels de hauteur (**Height**). La propriété **Scrollbars** peut être fixée à **ssAutoVertical**.

Cliquez sur les trois points en regard de la propriété **Columns** : un assistant va vous aider à créer les différentes colonnes du DBGrid. Cliquez chaque fois sur le bouton « + » pour ajouter une colonne et éditez les propriétés de celle-ci dans l'inspecteur d'objets.



Tous les titres (propriété **Title/Alignment**) étant centrés (**taCenter**), voici les propriétés des différentes colonnes à ajouter :

FieldName	Caption	Width + MaxSize	Alignment
Plaque	Plaque	79	taCenter
Marque	Marque	80	taLeftJustify
Modele	Modèle	110	TaLeftJustify
Cylindree	Cyl.	45	taCenter
Transmission	Trans.	45	taCenter
Prix	€/jour	45	taCenter

Pour terminer les réglages, il faut modifier les propriétés suivantes :

Propriété	Valeur
Options/dgIndicator	False
FixedCols	0

Comme nous l'avons mentionné juste avant de commencer le dépôt des composants sur la fiche, nous terminons par la propriété **DataSource** du DBGrid, que nous assignons à **DataModule1.DataSourceVoitures**.

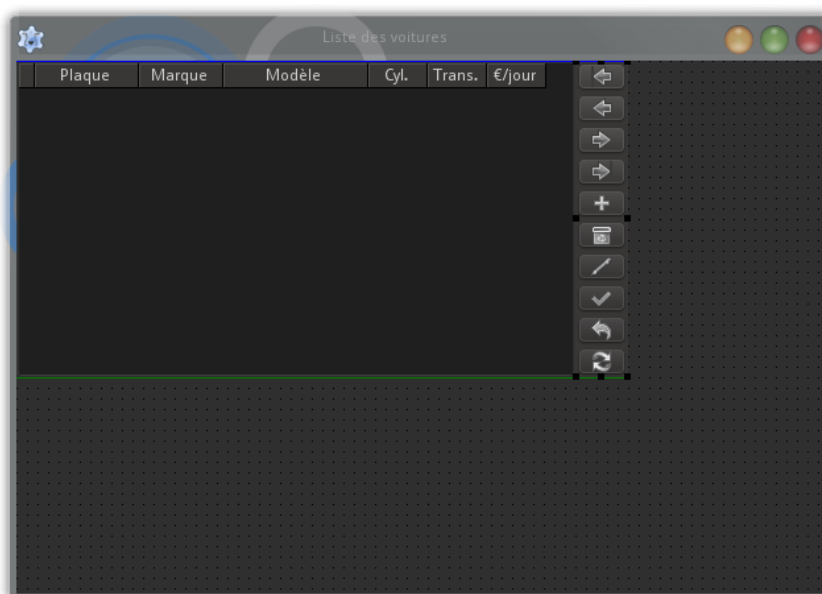


*Nous pourrions penser que le fait de faire un lien entre un composant de la fiche et un datasource qui se trouve dans un datamodule entraînerait de la part de Lazarus la déclaration de ce datamodule dans la clause **uses** de l'unité de la fiche. Il n'en est rien : c'est à nous de le faire. Ajoutez donc l'unité **DataAccess** à la clause **uses** de l'unité **Voitures**.*

## V-C-2 - TDBNavigator

Nous restons pour l'instant en terrain connu, car nous allons déposer sur la fiche un composant [TDBNavigator](#). C'est lui qui s'occupera de la navigation dans la table et de toutes les modifications de données.

Déposez-le à droite du DBGrid, nommez-le [dbnVoitures](#) (propriété [Name](#)), changez sa propriété [Direction](#) en [nbdVertical](#) (pour qu'il s'affiche verticalement), réglez sa largeur et sa hauteur pour qu'il vienne se coller le long de la bordure de droite du DBGrid :

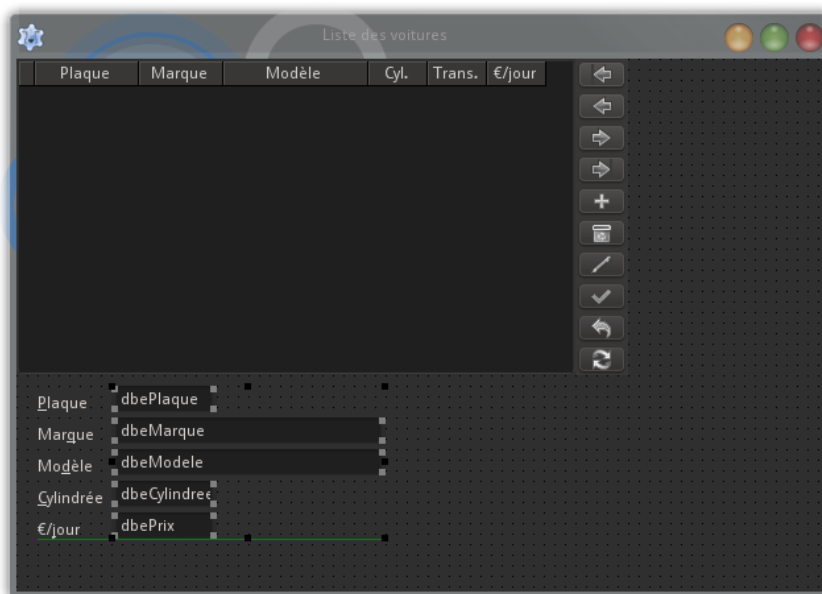


Fixez enfin sa propriété [DataSource](#) à [DataModule1.DataSourceVoitures](#).

## V-C-3 - TDBEdit

Nous allons à présent (« enfin », direz-vous peut-être) découvrir un nouveau composant spécialisé : le [TDBEdit](#). Il s'agit, comme son nom l'indique, d'un champ d'édition qui sera lié à un champ d'une table de base de données. Nous allons en déposer plusieurs, en dessous du DBGrid, accompagnés de classiques [TLabels](#) (de l'onglet [Standard](#)) :





Leurs propriétés respectives **Name** et **Caption** (pour les **TLabel**), et **Name** et **DataField** (pour les **TDBEdit**) seront fixées comme suit :

Name (TLabel)	Caption (TLabel)	Name (TDBEdit)	DataField (TDBEdit)
lblPlaque	&Plaque	dbePlaque	Plaque
lblMarque	Mar&que	dbeMarque	Marque
lblModele	Mo&dèle	dbeModele	Modele
lblCylindree	&Cylindrée	dbeCylindree	Cylindree
lblPrix	€/&jour	dbePrix	Prix

Élargissez les deux **TDBEdit** correspondant à la marque et au modèle de voiture, pour laisser suffisamment de place au texte qui s'y placera.

Sélectionnez les cinq **TDBEdit**, en pressant la touche **Shift** tout en cliquant sur les composants, et assignez d'un seul coup **DataModule1.DataSourceVoitures** à leur propriété **DataSource**.

#### V-C-4 - TDBRadioGroup

Le champ **Transmission** de la table **Voitures** indique si la voiture est équipée d'une transmission automatique ou manuelle ; ce champ est destiné à recevoir la valeur « A » ou « M ». Très naturellement, nous allons confier cette alternative à deux boutons radio, au sein d'un composant **TDBRadioGroup**.

Depuis l'onglet **Data Controls** (toujours le même), déposez à droite, en dessous du DBGrid, un composant **TDBRadioGroup**, que vous renommez directement **dbrgTransmission**. Sa propriété **Caption** devient « Transmission » et sa taille 90 (**Width**) par 73 (**Height**). Fort logiquement, vous assignerez à la propriété **DataField** le nom de champ **Transmission**.

Dans l'inspecteur d'objets, cliquez sur les trois points qui correspondent à la propriété **Items** du composant : un assistant va vous permettre d'énumérer les options qui seront proposées dans le groupe de boutons radio. Ce ne sera pas très long, puisque les valeurs possibles sont « A » et « M » :

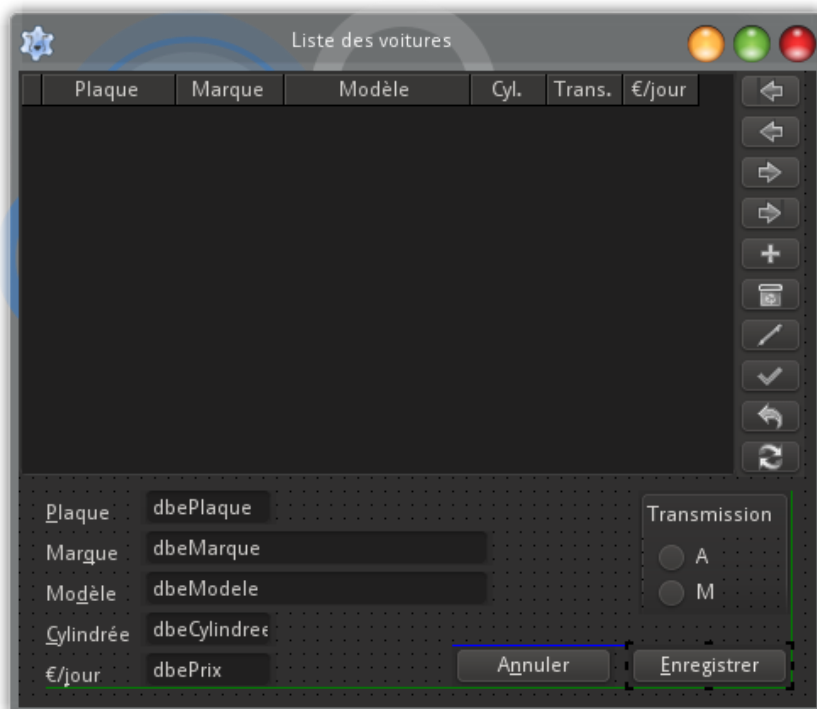


Nous terminons, comme chaque fois, par la propriété `DataSource` du composant, qui est initialisée elle aussi à `DataModule1.DataSourceVoitures`.

#### V-C-5 - Finalisation de la fiche

Nous avons terminé le dépôt de tous les composants spécialisés sur la fiche. Nous allons la finaliser et la tester.

Il reste de la place, en bas et à droite de la fiche, pour ajouter deux boutons classiques : le premier, que nous renommerons en `btnEnregistrer` et dont nous initialiserons la propriété `Caption` à « &Enregistrer », et le second, que nous appellerons `btnAnnuler` et dont la propriété `Caption` sera « A&nnuler ». Réglez la taille de la fiche pour obtenir quelque chose de joli :



Le but de notre fiche sera de permettre à l'utilisateur de modifier le contenu de la table [Voitures](#) de la base de données. Dès que notre dialogue s'affichera, toutes les voitures devront être chargées dans le DBGrid. Vous connaissez à présent la requête SQL qui permet de le faire :

```
SELECT * FROM Voitures;
```

Mais ! Rappelez-vous, nous avons pris le parti, dans cette application, de regrouper toute l'interface avec la base de données dans le *datamodule*. Nous n'allons donc pas implémenter notre requête dans le code de notre fiche, mais bien dans celui du *datamodule*.

Dans l'inspecteur d'objets, sélectionnez la fiche [CarForm](#) elle-même, cliquez sur l'onglet [Événements](#), puis sur les trois points en regard de l'événement [OnShow](#). Complétez la nouvelle méthode créée par Lazarus comme ceci :

```
procedure TCarForm.FormShow(Sender: TObject);
(* Chargement de la liste des voitures *)
begin
    DataModule1.ChargementVoitures;
end;
```

Nous confions donc le chargement de la table à une méthode [ChargementVoitures](#), que nous n'avons pas encore écrite dans le *datamodule*. Nous le ferons juste après.

Les deux boutons que nous avons ajoutés en dernier vont permettre à l'utilisateur d'enregistrer les modifications, ou bien de quitter le dialogue sans les enregistrer. Nous allons créer une propriété [Enregistre](#) (nous aimerions écrire « Enregistré » mais les caractères accentués ne sont pas - pas encore ? - autorisés dans la syntaxe du Pascal) pour notre fiche, de type booléen, qui va permettre de savoir si les données ont bien été enregistrées au moment de fermer le dialogue.

Créez un champ [FEnregistre](#) dans la section [strict private](#) de la fiche, ainsi que la propriété dont nous venons de parler et son setter :

```
type
{ TCarForm }
TCarForm = class(TForm)
```

```

    btnEnregistrer: TButton;
    btnAnnuler: TButton;
    dbePlaque: TDBEdit;
    dbeMarque: TDBEdit;
    dbeModele: TDBEdit;
    dbeCylindree: TDBEdit;
    dbePrix: TDBEdit;
    dbgVoitures: TDBGrid;
    DBNavigator1: TDBNavigator;
    dbrgTransmission: TDBRadioGroup;
    lblPlaque: TLabel;
    lblMarque: TLabel;
    lblModele: TLabel;
    lblCylindree: TLabel;
    lblPrix: TLabel;
    procedure FormShow(Sender: TObject);
    // DÉBUT DE L'AJOUT
strict private
    FEnregistre : Boolean;
private
    procedure SetEnregistre (AValue : Boolean);
public
    property Enregistre : Boolean read FEnregistre write SetEnregistre;
    // FIN DE L'AJOUT
end;

```

N'oublions pas d'initialiser cette propriété à **False** dès l'affichage du dialogue :

```

procedure TCarForm.FormShow(Sender: TObject);
(* Chargement de la liste des voitures *)
begin
    // AJOUT :
    Enregistre := False;
    // FIN AJOUT
    DataModule1.ChargementVoitures;
end;

```

Voici le code du setter :

```

procedure TCarForm.SetEnregistre (AValue : Boolean);
(* Setter de la propriété Enregistre *)
begin
    if FEnregistre = AValue
    then
        Exit;
    FEnregistre := AValue;
end;

```

Faisons en sorte que l'utilisateur reçoive un message de confirmation, s'il veut fermer le dialogue sans que les données soient enregistrées. Cela se fera en réponse à l'événement **OnCloseQuery** (cliquez sur les trois points en regard de cet événement, dans l'inspecteur d'objets) :

```

procedure TCarForm.FormCloseQuery (Sender : TObject; var CanClose : Boolean);
(* Demande éventuelle de confirmation de fermeture sans enregistrer *)
begin
    if Enregistre
    then
        CanClose := True
    else
        CanClose := (MessageDlg('Voulez-vous fermer sans enregistrer ?', mtConfirmation, [mbYes,
        mbNo], 0) = mrYes);
end;

```

Il nous reste juste à implémenter les méthodes qui vont réagir à un clic sur les boutons « Enregistrer » et « Annuler ». Cliquez successivement sur les trois points qui correspondent à l'événement **OnClick** des deux boutons, et complétez les méthodes comme ceci :

```

procedure TCarForm.btnEnregistrerClick(Sender: TObject);
(* Enregistre les modifications dans la base de données *)
begin
  Enregistre := DataModule1.SauvegardeVoitures;
  Close;
end;
procedure TCarForm.btnAnnulerClick(Sender: TObject);
(* Ferme la fenêtre sans enregistrer *)
begin
  Close;
end;

```

Vous le voyez, nous allons également tout de suite devoir écrire une méthode [SauvegardeVoitures](#) dans le *datamodule*.

## V-C-6 - Méthodes de chargement et de sauvegarde des données dans le datamodule

Allons-y, dans notre *datamodule*, et créons-y les deux méthodes publiques dont nous avons besoin. Il nous faudra une troisième méthode privée, que nous appellerons [Commit](#), qui sera chargée d'enregistrer toutes les modifications définitivement dans la base de données.

```

type
{ TDataModule1 }
TDataModule1 = class(TDataModule)
  DataSourceVoitures: TDataSource;
  SQLConnection: TMySQL56Connection;
  SQLQueryVoitures: TSQLQuery;
  SQLTransaction: TSQLTransaction;
private
  // AJOUT :
  function Commit : Boolean;
  // FIN AJOUT
public
  function Login : Boolean;
  procedure Logoff;
  // AJOUT :
  procedure ChargementVoitures;
  function SauvegardeVoitures : Boolean;
  // FIN AJOUT
end;

```

Après un **Shift-Ctrl-C**, voici le code à implémenter :

```

function TDataModule1.Commit: Boolean;
(* Sauvegarde des changements dans la base de données *)
begin
  Result := True;
  if SQLTransaction.Active
  then
    try
      SQLTransaction.Commit;
    except
      on e: ESQLException do
        begin (* Erreur renvoyée par MySQL *)
          MessageDlg('Erreur n° ' +
            IntToStr(e.ErrorCode) + ' : ' + e.Message,
            mtError, [mbOk], 0);
          Result := False;
        end;
      on e: EDatabaseError do
        begin (* Erreur générale *)
          MessageDlg('Erreur de sauvegarde des données', mtError, [mbOk], 0);
          Result := False;
        end;
      end;
    else
      Result := False;
    end;

```

```
end;
procedure TDataModule1.ChargementVoitures;
(* Chargement des voitures *)
begin
    with SQLQueryVoitures do
        begin
            Close;
            SQL.Text := 'SELECT * FROM Voitures;';
            Open;
        end;
    end;
end;
function TDataModule1.SauvegardeVoitures: Boolean;
(* Sauvegarde de la table Voitures *)
begin
    SQLQueryVoitures.ApplyUpdates;
    Result := Commit;
end;
```

Il n'y a rien de nouveau, à ce niveau, par rapport aux deux premiers exemples du tutoriel.

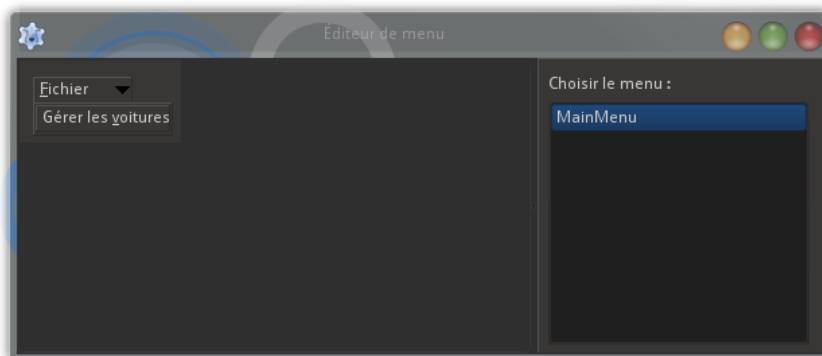
## V-C-7 - Test de la fiche

Vous êtes sans doute tout excité(e) à l'idée de tester le dialogue de modification des voitures que vous avez créé.

Retournez dans l'unité de la fiche principale. On pourrait trouver mille et un événements pour afficher le dialogue (un bouton sur la fiche principale, une réponse à un clic de souris sur la fenêtre, etc.). J'ai opté pour un menu.

Dans l'onglet **Standard**, choisissez un **TMainMenu** et déposez-le sur la fiche. Renommez-le éventuellement **MainMenu** (propriété **Name**). Cliquez sur les trois points à droite de sa propriété **Items**, pour ouvrir l'assistant de conception.

Un seul item est présent dans l'éditeur de menu. Dans l'inspecteur d'objets, changez sa propriété **Name** en **mnuFichier** et sa propriété **Caption** en « &Fichier ». Faites un clic droit sur l'item dans l'assistant, puis choisissez **Créer un sous-menu**. Cliquez sur le nouvel item créé et, dans l'inspecteur d'objets renommez-le **mnuFichierVoitures**, avec comme caption « Gérer les &voitures ».



Une fois que c'est fait, fermez l'assistant. Dans l'inspecteur d'objets, sélectionnez le tout dernier item qui vient d'être créé ; dans l'onglet **Événements**, cliquez sur les trois points à droite de l'événement **OnClick** et complétez la nouvelle méthode créée :

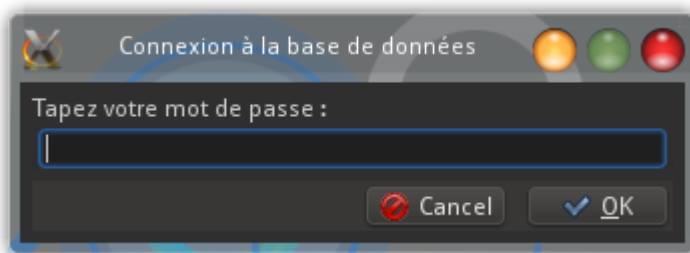
```
procedure TMainForm.mnuFichierVoituresClick(Sender: TObject);
(* Gestion de la liste des voitures *)
var
    LCarForm : TCarForm; (* Dialogue de gestion des voitures *)
begin
    LCarForm := TCarForm.Create(Self);
    try
        LCarForm.ShowModal;
```

```
finally
    FreeAndNil(LCarForm);
end;
end;
```

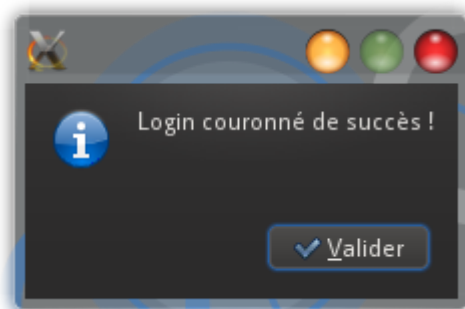
N'oubliez pas d'ajouter l'unité **Voitures** à la clause **uses** de l'unité **Main**.

Cette fois, nous sommes prêts : compilez et exécutez l'application.

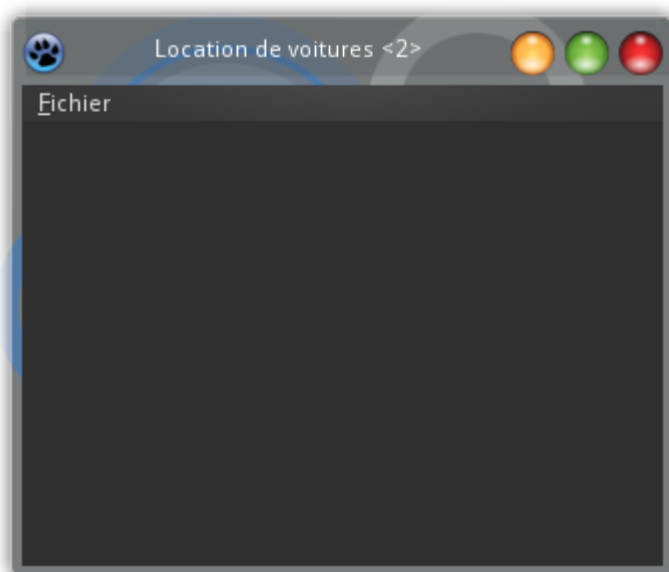
Entrez le mot de passe dans le premier dialogue :



Le petit message nous informe que la connexion à la base de données est couronnée de succès :



La fenêtre principale est bien vide pour l'instant (nous allons rapidement la remplir), juste le menu principal :



Dans le menu [Fichier](#), vous choisissez l'item [Gérer les voitures](#). La fiche que nous voulons tester apparaît :

Plaque	Marque	Modèle	Cyl.	Trans.	€/jour
AB-555-RB	Citroën	C3	1600	M	20
AB-612-BV	BMW	135i Coupé	3000	A	30
AC-811-CK	Citroën	C3	1600	M	20
AM-398-ER	Daihatsu	Sirion	1300	M	17.25
AM-400-SU	Ford	Focus ST	2000	M	23
AM-436-FD	Ford	Taurus SHO	3500	A	36.25
BN-101-AR	Citroën	C3	1600	M	20
BU-425-GH	Citroën	C4 Coupe 2.0VTS	2000	A	27.5
CB-135-RK	Fiat	500 1.2 8V Lounge	1200	M	17
CU-004-MP	Volkswagen	New Beetle	2000	M	27.5

Plaque: AB-555-RB  
 Marque: Citro  
 Modèle: C3  
 Cylindrée: 1600  
 €/jour: 20

Transmission:  
☐ A  
☒ M

Annuler Enregistrer

Parcourez le DBGrid et voyez comment tous les contrôles que nous avons déposés sur la fiche se mettent à jour automatiquement. Faites l'expérience d'un modifier un, cliquez sur la petite coche (« Post ») du DBNavigator et voyez comment le champ se met à jour dans le DBGrid. Faites ce que vous voulez ; pensez simplement que lorsque vous aurez cliqué sur [Enregistrer](#), vos modifications seront injectées dans la base de données.

## V-C-8 - Exercice : réaliser le dialogue de gestion des clients

Je vous propose un exercice, à ce stade : avec ce que vous venez de voir, vous devriez être capable de réaliser seul(e) le dialogue de gestion des clients, avec des composants spécialisés. Le principe est identique à celui des voitures.

Petites précisions :

- appelez votre fiche [CustomerForm](#) et l'unité [Clients](#) ;
- n'affichez pas toutes les colonnes dans le DBGrid, sous peine d'avoir une fiche très, très large. Vous pouvez vous contenter des noms et prénoms.

La solution (du moins, une solution possible) se trouvera dans le code complet du projet, que vous trouverez tout à la fin de ce tutoriel.

## V-D - Une interface pour définir les requêtes SQL

Dans le souci de bien structurer notre application, nous avons centralisé l'interfaçage avec la base de données dans un *datamodule*. Nous allons encore aller un cran plus loin, en regroupant tout ce qui a trait à la syntaxe SQL dans une unité séparée.

SQL (acronyme de *Structured Query Language*), est un langage normalisé, devenu pratiquement universel, permettant d'exploiter un très grand nombre de bases de données. Chaque système de gestion de bases de données, malheureusement, ajoute ça et là de petites variantes, ou développe des extensions propres qui complètent le langage SQL de base.



Si nous voulons que notre application puisse aisément être transposée à un autre SGBD (par exemple, de MySQL à SQLite), nous n'avons qu'à modifier la syntaxe de nos requêtes, et le reste de l'application, hormis le connecteur, pourra rester pratiquement inchangé.

Nous allons déclarer une **interface**, qui va contenir toutes les requêtes utiles pour notre application. Pour chaque nouvelle syntaxe, il faudra déclarer une classe particulière qui sera **obligée** d'implémenter toutes les requêtes définies dans l'interface.

Voyez, au sujet des interfaces, le [tutoriel de Laurent Dardenne](#) et [celui de Robin Valtot](#).

À l'aide du tout premier bouton de la barre d'outils de Lazarus, créez une nouvelle unité, que vous enregistrerez immédiatement sous le nom [sql.pas](#).

Dans la section [type](#), créez une interface [ISQLSyntax](#) :

```
type
  ISQLSyntax = interface
  end;
```

À l'aide de la combinaison de touches **Shift-Ctrl-G**, créez automatiquement un GUID :

```
type
  ISQLSyntax = interface
    ['{4AF51BFD-D53D-43F7-9A36-17E859D467CE}']
  end;
```

La seule requête SQL que nous ayons utilisée jusqu'à présent est celle qui sélectionne toutes les voitures dans la table [Voitures](#). Créez une première fonction [SelectionVoituresToutes](#) :

```
type
  { ISQLSyntax }

  ISQLSyntax = interface
    ['{238542C2-ADA2-46BC-9138-4D270BEB85D0}']
    function SelectionVoituresToutes : String;
    (* Requête de sélection de toutes les voitures *)
  end;
```

Pressez la combinaison de touches **Shift-Ctrl-C** : il ne se passe... rien. En effet, les méthodes déclarées dans l'interface sont uniquement implémentées dans une classe descendante.

Toute classe descendante de cette interface sera donc obligée d'implémenter la fonction [SelectionVoituresToutes](#). Nous allons créer une classe pour la syntaxe MySQL :

```
TMySQLSyntax = class(TInterfacedObject, ISQLSyntax)
  function SelectionVoituresToutes : String;
end;
```

Cette fois, pressez **Shift-Ctrl-C** et complétez la méthode dans la section implémentation :

```
function TMySQLSyntax.SelectionVoituresToutes : String;
  (* Requête de sélection de toutes les voitures *)
begin
  Result := 'SELECT * FROM Voitures;';
end;
```

Déclarez une variable de type [TMySQLSyntax](#) dans la partie interface de l'unité :

```
var
  SQLSyntax : TMySQLSyntax;  (* Syntaxe propre à MySQL *)
```



Pour bien illustrer que toute classe descendante de l'interface sera obligée d'implémenter toutes ses méthodes, faites l'expérience de créer une méthode **bidon** dans l'interface et de compiler : l'erreur renvoyée sera « No matching implementation for interface method 'bidon' found ».

Retournez dans l'unité [DataAccess](#) (le *datamodule*) et modifiez la méthode [ChargementVoitures](#) :

```
procedure TDataModule1.ChargementVoitures;
(* Chargement des voitures *)
begin
  with SQLQueryVoitures do
    begin
      Close;
      // DÉBUT DES MODIFICATIONS
      SQL.Text := MySQLSyntax.SelectionVoituresToutes;
      // FIN DES MODIFICATIONS
      Open;
    end;
  end;
end;
```

N'oubliez pas d'ajouter l'unité [SQL](#) à la clause [uses](#).

À présent, réfléchissons à l'endroit où nous allons instancier la classe [TMySQLSyntax](#). À quel moment en aurons-nous besoin ? À chaque fois qu'une commande SQL devra être générée, c'est-à-dire à peu près partout dans l'application. Donc le meilleur endroit est de la créer au moment de l'affichage de la fenêtre principale, après le login, puis de la libérer à la fermeture de l'application. Il faut donc modifier les méthodes [FormShow](#) et [FormClose](#) de la fiche principale (dans l'unité [Main](#)) :

```
procedure TMainForm.FormShow(Sender: TObject);
(* Demande de mot de passe *)
begin
  if DataModule1.Login
  then
    begin
      // DÉBUT DE L'AJOUT
      (* Initialisation de la classe de syntaxe SQL *)
      SQLSyntax := TMySQLSyntax.Create;
      // FIN DE L'AJOUT

      ShowMessage('Login couronné de succès !');
    end
  else
    Close;
  end;
procedure TMainForm.FormClose (Sender : TObject; var CloseAction : TCloseAction);
(* Fermeture propre de la connexion à la base de données *)
begin
  // DÉBUT DE L'AJOUT
  (* Libération des commandes SQL *)
  SQLSyntax.Free;
  // FIN DE L'AJOUT
  (* Déconnexion *)
  DataModule1.Logoff;
end;
```

Encore une fois, ajoutez l'unité [SQL](#) à la clause [uses](#) de l'unité [Main](#).

## V-E - Utilisation d'un TDBGrid sans TDBNavigator

Nous allons nous occuper de notre fenêtre principale, qui est bien vide pour l'instant. Elle contiendra la liste des locations, mais aussi différents filtres permettant de n'afficher que les locations répondant à différents critères.

### V-E-1 - TDBGrid

La liste des locations sera contenue dans un DBGrid, dont les colonnes seront les suivantes :

- les nom et prénom du client ;
- la plaque de la voiture ;
- la marque de la voiture ;
- son modèle ;
- la date de début de location ;
- la date de fin prévue ;
- la date de rentrée de la voiture à l'issue de la location.

J'ai décidé d'afficher le nom et le prénom du client dans une seule colonne. Pour ce faire, une solution est d'ajouter dans la table [Clients](#) un champ supplémentaire. En réalité, nous n'avons pas besoin de créer une colonne qui contiendra en permanence des données : nous allons créer une colonne **virtuelle**, dont le contenu sera **calculé** à partir des colonnes [Nom](#) et [Prénom](#) (qui ne peuvent être nulles, ainsi que nous les avons conçues).

Direction le navigateur web et phpMyAdmin : dans l'onglet [SQL](#), collez la commande suivante :

```
ALTER TABLE Clients ADD NomPrenom VARCHAR(81) AS ( CONCAT(Nom, ' ', Prenom) ) VIRTUAL;
```

Cette nouvelle colonne [NomPrenom](#) ne prendra aucune place dans la base de données, et nous pourrons nous en servir pour afficher ensemble le nom et le prénom de chaque client.

Direction le *datamodule* : cliquez sur l'onglet de l'unité [DataAccess](#) et pressez **F12**.

Déposez sur la fiche un nouveau composant [TSQLQuery](#) et un nouveau [TDataSource](#), que vous renommez tout de suite respectivement [SQLQueryMain](#) et [DataSourceMain](#). Vous connaissez la musique : assignez [SQLConnection](#) à la propriété [Database](#) du [SQLQuery](#) et [SQLQueryMain](#) à la propriété [DataSet](#) du [DataSource](#).

En cliquant sur les trois points en regard de la propriété [FieldDefs](#) du [SQLQuery](#), définissez les différents champs à l'aide de l'assistant :

Name	DataType	Size
IdLocation	ftWord	0
IdClient	ftWord	0
Plaque	ftString	12
DateDebut	ftDateTime	0
DateFin	ftDateTime	0
DateRentree	ftDateTime	0
Assurance	ftSmallInt	0
Plaque_1	ftString	12
Marque	ftString	20
Modele	ftString	20
Cylindree	ftInteger	0
Transmission	ftFixedChar	1
Prix	ftFloat	0
IdClient_1	ftWord	0
Nom	ftString	40
Prenom	ftString	40
CodePostal	ftString	10
Localite	ftString	50
Rue	ftString	80
Numero	ftString	10
Telephone	ftString	40
Email	ftString	50
NomPrenom	ftString	81

Êtes-vous surpris(e) par le nombre de champs ? C'est normal : la table [Locations](#) fait référence aux tables [Voitures](#) et [Clients](#), par ses clés étrangères, et dans toute requête de sélection nous ferons ce que l'on appelle une **jointure**. Laissez-moi le bénéfice du doute pendant quelques minutes encore.

Revenez à la fiche principale, en cliquant sur l'onglet [Main](#) et en pressant **F12**.

Élargissez franchement la fiche et déposez-y un composant [TDBGrid](#), que vous renommez [dbgMain](#) et dont vous fixez la taille à 784 x 312.

À l'aide de l'assistant de création de colonnes (que vous exécutez en cliquant sur les trois points en regard de la propriété [Columns](#), dans l'inspecteur d'objets), créez les colonnes suivantes :

Titre ( <a href="#">Title</a> )	Taille ( <a href="#">Width</a> )	Champ ( <a href="#">FieldName</a> )
Client	240	NomPrenom
Plaque	80	Plaque
Marque	80	Marque
Modèle	110	Modele
Début	80	DateDebut
Fin	80	DateFin
Rentrée	80	DateRentree

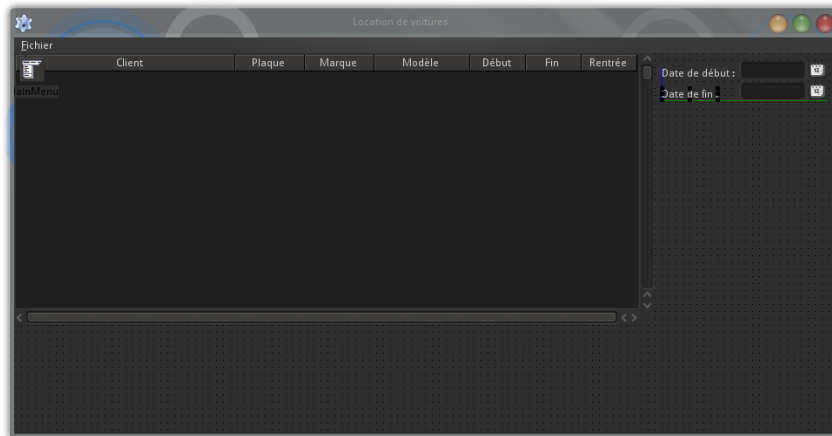
Pour obtenir un affichage correct dans les trois colonnes de dates, assignez à leur propriété [DisplayFormat](#) la valeur suivante : « dd"/"mm"/"yyyy ».

Pour terminer, affectez [DataModule1.DataSourceMain](#) à la propriété [DataSource](#) du DBGrid.

## V-E-2 - Filtres

Est-il intéressant d'afficher l'entièreté des locations, c'est-à-dire passées, présentes et futures ? Pas tellement, alors nous allons ajouter des **filtres** sur la fenêtre principale.

À droite du DBGrid, déposez deux **TLabel** (onglet **Standard**) et deux **TDateEdit** (onglet **Misc**) :



Renommez-les respectivement **lblFiltreDateDebut** et **lblFiltreDateFin**, pour les **TLabel**, et **deFiltreDateDebut** et **deFiltreDateFin** pour les **TDateEdit**. Fixez les propriétés **Caption** des labels à « Date de début : » et « Date de fin : ».

Il nous faut fixer le format des deux composants d'édition de date : dans l'inspecteur d'objets, leur propriété **DateOrder** doit être assignée à **doDMY**. Vous voyez que le contrôle d'édition lié est automatiquement configuré par Lazarus en « \_\_/\_\_/\_\_\_\_ », pour permettre d'afficher la date au format « jj/mm/aaaa » auquel nous sommes habitués.

Nous allons ajouter un autre filtre, en dessous des deux que nous venons de créer, permettant de n'afficher que les locations pour lesquelles une assurance complémentaire a été contractée par le client. Il s'agira d'une simple case à cocher (composant **TCheckBox**), que l'on trouve dans l'onglet **Standard**.

Assignez **cbFiltreAssurance** à sa propriété **Name** et « Assurance complémentaire » à sa propriété **Caption**.

Nous allons faire en sorte qu'au démarrage de l'application, les filtres sur la date de début et la date de fin soient fixés à un mois dans le passé et un mois dans le futur. Dans le code source de l'unité **Main**, ajoutez ce code à la méthode **FormShow** :

```
procedure TMainForm.FormShow(Sender: TObject);
(* Demande de mot de passe *)
begin
  if DataModule1.Login
  then
    begin
      (* Initialisation de la classe de syntaxe SQL *)
      SQLSyntax := TMySQLSyntax.Create;
      (* Initialisation des filtres de dates *)
      // DÉBUT DE L'AJOUT
      deFiltreDateDebut.Date := IncDay(Today, -30);
      deFiltreDateFin.Date := IncDay(Today, 30);
      // FIN DE L'AJOUT
    end
  else
    Close;
  end;
end;
```

Ajoutez également l'unité **DateUtils** à la clause **uses** de l'unité.

## V-E-3 - Requête SQL de sélection

Penchons-nous à présent sur la requête SQL qui va nous permettre de charger le DBGrid en tenant compte des filtres.

Dans l'unité [SQL](#), nous ajoutons une fonction à l'interface [ISQLSyntax](#) :

```
function SelectionLocationsFiltre (
    (* Requête de sélection de locations avec critères *)
    const ADateDebut, ADateFin : TDateTime;      (* Dates de début et de fin *)
    const AAssurance : Boolean                    (* Avec assurance complémentaire *)
) : String;
```

Comme prévu, nous passons comme paramètres à la fonction les dates de début et de fin, ainsi que l'option d'affichage de l'assurance complémentaire.

Ajoutez identiquement cette fonction à la classe [TMySQLSyntax](#) qui dérive de l'interface. À l'aide de la combinaison de touches habituelle **Shift-Control-C**, créez l'implémentation de cette fonction et complétez-la comme suit :

```
function TMySQLSyntax.SelectionLocationsFiltre (const ADateDebut, ADateFin : TDateTime;
                                                const AAssurance : Boolean) : String;
(* Requête de sélection de locations avec critères *)
begin
    Result := 'SELECT * FROM Locations' +
        ' INNER JOIN Voitures ON Locations.Plaque = Voitures.Plaque' +
        ' INNER JOIN Clients ON Locations.IdClient = Clients.IdClient' +
        ' WHERE DateDebut >= ''' + DateToStr(ADateDebut, FormatDate) + ''' +
        ' AND DateFin <= ''' + DateToStr(ADateFin, FormatDate) + ''' AND Assurance = ';

    if AAssurance
    then
        Result := Result + '1';
    else
        Result := Result + '0';
end;
```

Nous avons beaucoup de choses à dire à propos de cette requête.

Tout d'abord, je vous ai annoncé un peu plus haut que nous allions recourir à une **jointure**. Cette technique permet d'inclure à la requête différentes tables auxquelles il est fait référence, par le biais des clés étrangères, dans la table sur laquelle s'effectue la requête. C'est ainsi que la table [Locations](#) fait référence à la table [Clients](#) par son champ [IdClient](#), et à la table [Voitures](#) par son champ [Plaque](#). La requête de sélection joint donc la table [Voitures](#) par la commande SQL **INNER JOIN Voitures ON Locations.Plaque = Voitures.Plaque** et la table [Clients](#) par la commande **INNER JOIN Clients ON Locations.IdClient = Clients.IdClient**.

Ensuite, c'est dans la clause **WHERE** que nous effectuons les tests sur les différents filtres.

Une chose importante : il faut que les dates de début et de fin soient incluses dans la requête sous forme de texte, mais dans un format reconnu par MySQL.



Ce format de date n'a rien à voir avec le format [doDMY](#) que nous avons fixé pour les deux [TDateEdit](#).

Nous allons donc ajouter à l'interface [ISQLSyntax](#) une fonction [FormatDate](#) qui va retourner une structure de type [TFormatSettings](#) utilisable par la fonction de conversion [DateToStr](#) :

```
function FormatDate : TFormatSettings;
(* Format de date et le séparateur compatibles avec le SGBD *)
```

Si vous avez la curiosité de regarder la déclaration du type [TFormatSettings](#), vous verrez qu'il contient une vingtaine de champs. Seuls deux d'entre eux nous intéressent pour assurer la compatibilité du format de date avec MySQL :

- [DateSeparator](#) ;
- [ShortDateFormat](#).

Voici l'implémentation de la fonction [FormatDate](#) dans la classe [TMySQLSyntax](#) :

```
function TMySQLSyntax.FormatDate: TFormatSettings;
(* Formats de date et de séparateur compatibles avec le SGBD *)
begin
    Result.DateSeparator := '-';
    Result.ShortDateFormat := 'yyyy-mm-dd';
end;
```

Vous voyez que la base de données est configurée pour travailler avec un format de date anglo-saxon.

Il reste un petit détail à mentionner, à propos du filtre sur l'assurance complémentaire : le booléen passé en paramètre doit être transformé en valeur 0 ou 1.

## V-E-4 - Chargement du DBGrid

Nous y sommes presque : le DBGrid est prêt, la requête SQL de sélection est écrite, il ne nous reste plus qu'à créer une méthode qui va charger le DBGrid.

Pour respecter notre logique, cette méthode se trouvera dans le *datamodule*. Hop, un clic sur l'onglet [DataAccess](#) dans l'éditeur de source !

Dans la section [public](#) du *datamodule*, créez la procédure suivante :

```
procedure ChargementLocations (const Requete : String);
```

Voici son implémentation :

```
procedure TDataModule1.ChargementLocations (const Requete: String);
(* Charge la table des locations *)
begin
    SQLQueryMain.Close;
    SQLQueryMain.SQL.Text := Requete;
    SQLQueryMain.Open;
end;
```

La requête passée comme paramètre sera celle que nous venons de créer.

Retournez dans l'unité [Main](#) et ajoutez à la méthode [FormShow](#) :

```
procedure TMainForm.FormShow(Sender: TObject);
(* Demande de mot de passe *)
begin
    if DataModule1.Login
    then
        begin
            (* Initialisation de la classe de syntaxe SQL *)
            SQLSyntax := TMySQLSyntax.Create;
            (* Initialisation des filtres de dates *)
            deFiltreDateDebut.Date := IncDay(Today, -30);
            deFiltreDateFin.Date := IncDay(Today, 30);
            // DÉBUT DE L'AJOUT
            (* Chargement de la liste des locations *)
```

```

        DataModule1.ChargementLocations(SQLSyntax.SelectionLocationsFiltre(deFiltreDateDebut.Date,
deFiltreDateFin.Date, cbFiltreAssurance.Checked));
        // FIN DE L'AJOUT
    end
else
    Close;
end;
end;

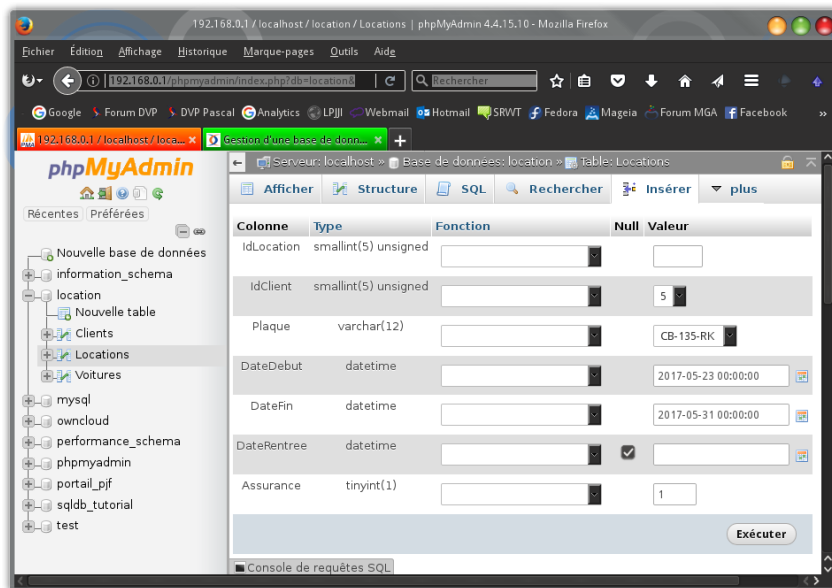
```

De cette manière, dès l'apparition de la fenêtre principale de l'application, s'affichera la liste des locations répondant aux filtres par défaut.

Je suis sûr que vous avez envie de tester votre programme. Oui, mais ! Il n'y a encore aucune location dans la base de données, donc le DBGrid sera vide. Faisons donc l'exercice de créer quelques locations directement dans la base, à l'aide de phpMyAdmin. Ce faisant, nous visualiserons, d'ailleurs, la jointure avec les voitures et les clients.

Rendez-vous dans votre navigateur et connectez-vous à la base de données. Sélectionnez, dans la colonne de gauche de phpMyAdmin, la table **Locations**, et cliquez à droite, sur l'onglet **Insérer**.

Laissez vide la première colonne, **IdLocation** : il s'agit d'un index qui sera automatiquement incrémenté. Dans la liste déroulante du champ **IdClient**, choisissez l'identificateur d'un client ; dans celle du champ **Plaque**, choisissez l'un des véhicules. Ces deux valeurs font le lien avec les deux autres tables, c'est à ce niveau que s'effectue la jointure. Définissez également une date de début et de fin (choisissez des dates proches d'aujourd'hui, car rappelez-vous, par défaut les filtres sont réglés à un mois dans le passé et un mois dans le futur), et inscrivez 0 ou 1 comme valeur pour le champ **Assurance** :

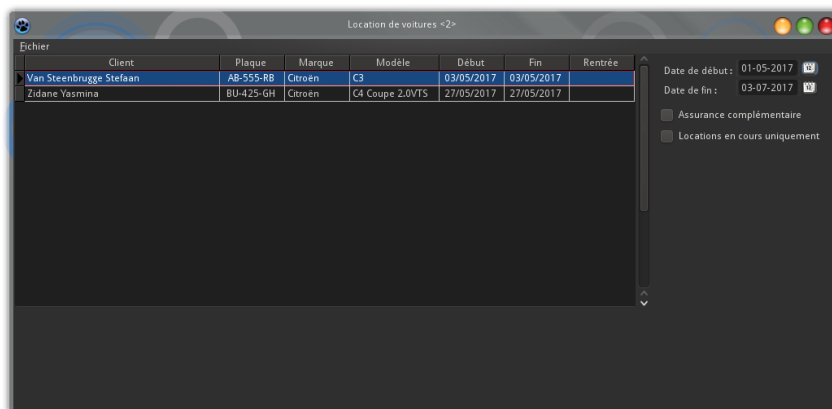


Cliquez sur le bouton **Exécuter** pour créer la location.

Vous pouvez répéter l'opération une ou deux fois, histoire d'avoir quelque chose à afficher dans le DBGrid de l'application.

À présent, vous êtes prêt(e) à exécuter votre programme. Allez-y !





Yes !

Réfléchissons encore un peu avant d'aller fêter cela. À chaque fois qu'une modification va être apportée à l'un des filtres, il faudra actualiser le contenu du DBGrid. Par conséquent, nous devons exécuter la méthode `DataModule1.SelectionLocationsFiltre` en réponse à tout événement `OnChange` d'un filtre.

Ce sera vite fait. Sélectionnez successivement les deux `TDateEdit` et le `TCheckBox` ; dans l'inspecteur d'objets, cliquez sur l'onglet `Événements` puis sur les trois points correspondant à l'événement `OnChange`. Complétez les trois méthodes événementielles créées par le même code :

```
procedure TMainForm.deFiltreDateDebutChange(Sender: TObject);
(* Modification au filtre : mise à jour du contenu du DBGrid *)
begin
    DataModule1.ChargementLocations(SQLSyntax.SelectionLocationsFiltre(deFiltreDateDebut.Date,
    deFiltreDateFin.Date, cbFiltreAssurance.Checked));
end;
procedure TMainForm.cbFiltreAssuranceChange(Sender: TObject);
(* Modification au filtre : mise à jour du contenu du DBGrid *)
begin
    DataModule1.ChargementLocations(SQLSyntax.SelectionLocationsFiltre(deFiltreDateDebut.Date,
    deFiltreDateFin.Date, cbFiltreAssurance.Checked));
end;
procedure TMainForm.deFiltreDateFinChange(Sender: TObject);
(* Modification au filtre : mise à jour du contenu du DBGrid *)
begin
    DataModule1.ChargementLocations(SQLSyntax.SelectionLocationsFiltre(deFiltreDateDebut.Date,
    deFiltreDateFin.Date, cbFiltreAssurance.Checked));
end;
```

## V-E-5 - Exercice : ajouter un filtre pour n'afficher que les locations en cours

Je vous propose comme exercice d'ajouter un filtre pour que seules les locations en cours (dont la date de rentrée de la voiture est vide) s'affichent dans le DBGrid. Utilisez une case à cocher et réfléchissez bien à ce qu'il faut modifier dans la requête SQL.



*Une valeur NULL ne se compare pas comme n'importe quelle valeur. Pour la tester, on peut utiliser la syntaxe `Champ IS NULL` ou `Champ IS NOT NULL`.*

Une proposition de solution se trouve dans le code source complet de l'application.

## V-E-6 - Des boutons classiques pour l'ajout, la modification et la suppression

Nous avons pris le parti de ne pas utiliser de [TDBNavigator](#), et donc d'utiliser le DBGrid comme une *listbox* classique. Nous allons par conséquent substituer des boutons normaux à ceux du DBNavigator.

En dessous des filtres, ajoutez trois composants de type [TButton](#) (onglet [Standard](#)) :

Nom ( <a href="#">Name</a> )	Libellé ( <a href="#">Caption</a> )
btnAjouter	&Nouvelle location
btnModifier	&Modifier la location
btnSupprimer	&Supprimer la location

Afin que le DBGrid se comporte comme une simple *listbox*, nous devons y désactiver les possibilités d'édition des données et faire en sorte que la sélection porte sur une ligne entière (et non plus sur une seule cellule). Sélectionnez-le et réglez les propriétés suivantes dans l'inspecteur d'objets :

Propriété	Valeur
ReadOnly	True
Options / dgDisableDelete	True
Options / dgDisableInsert	True
Options / dgEditing	False
Options / dgRowHighlight	True
Options / dgRowSelect	True

Maintenant, vous pouvez aller faire une pause bien méritée. Rechargez bien vos batteries et ne buvez pas trop, car ce qui va suivre va nécessiter toute votre attention.

## V-F - Utilisation de composants non spécialisés

Créez une nouvelle fiche à l'aide du second bouton de la barre d'outils de Lazarus. Remplacez son nom [Form1](#) par [NewLeasingForm](#), ce qui transformera automatiquement son type en [TNewLeasingForm](#). Assignez « Location » à sa propriété [Caption](#). Tant que vous y êtes, donnez-lui comme dimensions 508 pixels ([Width](#)) sur 181 pixels ([Height](#)).

Cette fiche constituera le dialogue de création d'une nouvelle location. Comme nous sommes prévoyants, nous créerons la fiche de manière à ce qu'elle puisse facilement être utilisée pour modifier une location existante, dans une classe descendante.

Enregistrez-la et donnez à la nouvelle unité le nom [Locations](#).

Nous allons déposer sur la fiche une série de contrôles classiques, c'est-à-dire non spécialisés dans les bases de données.

À partir du coin supérieur gauche, déposez quatre [TLabel](#) l'un en dessous de l'autre :

<a href="#">Name</a>	<a href="#">Caption</a>
lblClient	Client :
lblVoiture	Voiture :
lblDateDebut	Date de début :
lblDateFin	Date de fin :

En regard de ces quatre labels, déposez les quatre composants suivants :

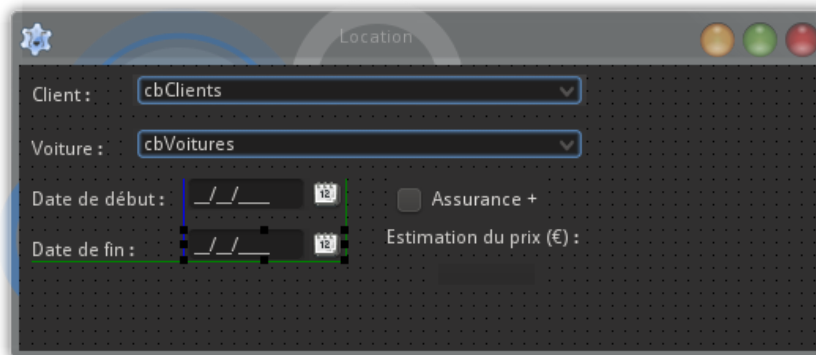
- deux **TComboBox** (onglet **Standard**), respectivement nommés (propriété **Name**) **cbClients** et **cbVoitures**, de largeur 288 ;
- deux **TDateEdit** (onglet **Misc**), respectivement nommés **deDateDebut** et **deDateFin**.

Les deux **TComboBox** contiendront la liste des clients et des voitures. Les deux **TDateEdit**, eux, permettront de définir les dates de début et de fin de location. Fixez leur propriété **DateOrder** à **doDMY** et cochez leur propriété **DefaultToday** : ainsi, par défaut ils contiendront la date du jour.

À droite des **TDateEdit**, déposez un **TCheckBox** (onglet **Standard**), que vous renommez **cbAssurance** et dont vous initialisez la propriété **Caption** à « Assurance + ».

En dessous de cette case à cocher, déposez un autre **TLabel** nommé **lblEstimationPrix**, contenant « Estimation du prix (€) : » dans sa propriété **Caption**, et enfin un **TStaticText** (de l'onglet **Additional**), nommé **stEstimationPrix**, dont vous effacez la propriété **Caption** et vous fixez la propriété **Alignement** à **taCenter**.

La fiche devrait ressembler à ceci :



Dans l'espace libre à droite, nous allons déposer divers contrôles permettant de filtrer les voitures qui se trouveront dans la liste.

D'abord, au milieu de l'espace vide, un **TLabel**, nommé **lblCylindree**, qui affiche « Cylindrée », et dont la propriété **Font.Style** est initialisée à **[fsUnderline]** (pour souligner le texte).

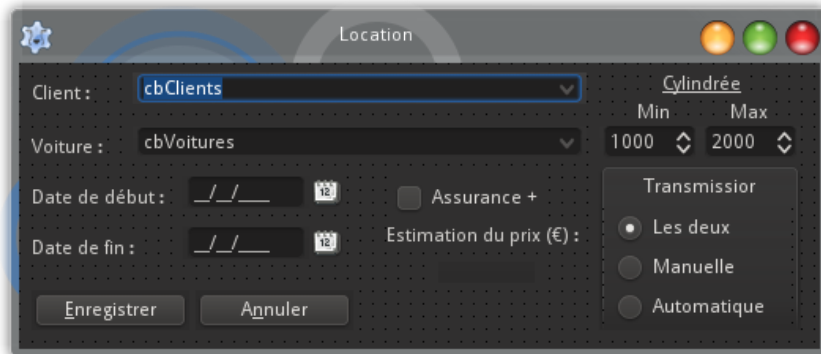
Ensuite, deux **TLabel** l'un à côté de l'autre, nommés **lblCylindreeMin** et **lblCylindreeMax**, dont les propriétés **Caption** sont respectivement « Min » et « Max ». En dessous de ces labels, deux **TSpinEdit** (onglet **Misc**), respectivement nommés **seCylindreeMin** et **seCylindreeMax**. Ces deux composants permettront de fixer les limites inférieure et supérieure de cylindrée des voitures de la liste. Il faut définir leurs bornes minimale et maximale (propriétés **MinValue** et **MaxValue** : **500** et **9900** pour le premier, **600** et **10000** pour le second), ainsi que la valeur qui sera incrémentée ou décrétementée à chaque pas, la propriété **Increment**, que nous fixons à **100**. Par défaut, nous leur assignons comme valeur de départ **1000** et **2000**, dans leur propriété **Value**. La largeur des deux **TSpinEdit** sera fixée à 64.

Nous allons remplir l'espace restant à droite avec un filtre sur la transmission. Dans la palette **Standard**, sélectionnez un **TRadioGroup** et déposez-en un sous les **TSpinEdit**, en adaptant sa largeur et sa hauteur pour occuper l'espace libre tout en restant aligné avec les autres contrôles (autant que ça soit joli, n'est-ce pas !). Renommez-le **rgTransmission**.

Nous déposons dans le **TRadioGroup** trois boutons radio (**TRadioButton**, de l'onglet **Standard**), dont Lazarus fera automatiquement un groupe. Les trois boutons sont nommés **rbTransmissionX**, **rbTransmissionM** et **rbTransmissionA**, et leurs propriétés **Caption** sont assignées à « Les deux », « Manuelle » et « Automatique ». Vous devinez aisément que le premier bouton radio ne filtrera pas les voitures sur le critère de la transmission, et que les deux autres filtreront les boîtes manuelles ou automatiques. Si Lazarus ne l'a pas fait automatiquement, cochez la propriété **Checked** du premier bouton radio, afin qu'il soit sélectionné par défaut.

Il ne reste plus que deux boutons à placer, en bas et à gauche : deux **TButton** respectivement nommés **btnEnregistrer** et **btnAnnuler**, avec comme propriété **Caption** « &Enregistrer » et « A&nnuler ».

Nous avons terminé la conception de notre fiche :



Vous avez sûrement hâte d'afficher le nouveau dialogue, même vide. Retournez dans la fiche principale (unité **Main**), pressez **F12**, sélectionnez le bouton **btnAjouter** dans l'inspecteur d'objets, allez dans l'onglet **Événements** et cliquez sur les trois points pour créer une méthode qui répondra à l'événement **OnClick**. Ajoutez l'unité **Locations** à la clause **uses** de l'unité **Main**.

Sans surprise, voici le code d'affichage du dialogue :

```
procedure TMainForm.btnAjouterClick (Sender : TObject);
(* Ajout d'une nouvelle location *)
var
  LNewLeasingForm : TNewLeasingForm; (* Dialogue de collecte des données *)
begin
  LNewLeasingForm := TNewLeasingForm.Create(Self);
  try
    LNewLeasingForm.ShowModal;
  finally
    FreeAndNil (LNewLeasingForm);
  end;
  (* Mise à jour de la liste des locations affichée *)
  DataModule1.ChargementLocations (SQLSyntax.SelectionLocationsFiltre (deFiltreDateDebut.Date,
deFiltreDateFin.Date,
                                                                    cbFiltreAssurance.Checked,
cbFiltreEnCours.Checked));
```

Le dialogue ne contient aucun composant orienté bases de données, c'était le but que nous nous étions fixé. Comment alors faire le lien avec la base de données ?

## V-F-1 - Chargement des clients et des voitures dans des TComboBox

Pour la liste des clients et des voitures, nous allons utiliser une fonctionnalité bien pratique des *comboboxes* : à chaque élément de la liste peut être attaché un objet quelconque. Considérons la déclaration de la méthode **AddItem**, qui est héritée de la classe **TCustomComboBox** :

```
procedure TCustomComboBox.AddItem (
  const Item: String;
  AnObject: TObject
); virtual;
```

Le paramètre **Item** contient la chaîne de caractères à afficher et **AnObject** sera un objet lié à l'élément, qui pourra contenir des données supplémentaires.

Réfléchissons : en plus de son nom, qui va être affiché dans la *combobox* des clients, de quelles données supplémentaires avons-nous besoin pour un client ? De pas grand-chose, en fait, juste son identificateur dans la table **Clients** de la base de données (la clé primaire de la table). Et pour une voiture ? Là, c'est plus compliqué, car il y a différentes caractéristiques d'une voiture qui sont concernées par les filtres de notre boîte de dialogue : la cylindrée et la transmission. Le prix par jour est également requis pour le calcul de l'estimation du coût de la location. Et nous aurons également besoin de l'identificateur de la voiture dans la table **Voitures**, qui est sa plaque d'immatriculation.

Dans l'unité **DataAccess**, nous créons deux classes **TCBClient** et **TCBVoiture** (« CB » faisant référence aux *comboboxes* auxquelles elles sont destinées) :

```
type
{ TCBClient }

TCBClient = class
  (* Données invisibles d'un élément de combobox contenant la liste des clients *)
  strict private
    FIdClient : Integer;
  public
    property IdClient : Integer read FIdClient;
    constructor Create (const AIdClient : Integer);
end;
{ TCBVoiture }
TCBVoiture = class
  (* Données invisibles d'un élément de combobox contenant la liste des voitures *)
  strict private
    FPlaque : String;
    FCylindree : Integer;
    FTransmission : Char;
    FPrix : Real;
  public
    property Plaque : String read FPlaque;
    property Cylindree : Integer read FCylindree;
    property Transmission : Char read FTransmission;
    property Prix : Real read FPrix;
    constructor Create (const APlaque : String;
                        const ACylindree : Integer;
                        const ATransmission : Char;
                        const APrix : Real);
end;
```

Voici le code des deux méthodes **Create** :

```
constructor TCBClient.Create (const AIdClient : Integer);
(* Initialisation des champs *)
begin
  FIdClient := AIdClient;
end;
constructor TCBVoiture.Create (const APlaque : String;
                               const ACylindree : Integer;
                               const ATransmission : Char;
                               const APrix : Real);
(* Initialisation des champs *)
begin
  FPlaque := APlaque;
  FCylindree := ACylindree;
  FTransmission := ATransmission;
  FPrix := APrix;
end;
```

Nous confions au *datamodule* le soin de charger les clients et les voitures dans les *comboboxes* correspondantes. Dans sa section **public**, créez les deux méthodes suivantes :

```
procedure ChargementCBClients (var ComboBox : TComboBox);
procedure ChargementCBVoitures (var ComboBox : TComboBox;
                                const Requete : String);
```

Grâce à la combinaison de touches **Shift-Ctrl-C**, créez leur implémentation et complétez le code comme suit :

```
procedure TDataModule1.ChargementCBClients (var ComboBox : TComboBox);
(* Remplit une combobox avec les noms et prénoms des clients.
   Chaque élément est constitué d'un texte visible et de données invisibles *)
var
  LNomPrenom : String; (* Texte visible d'un élément *)
begin
  ChargementClients(SQLSyntax.SelectionClientsTous);
  with SQLQueryClients do
    while not EOF do
      begin
        LNomPrenom := FieldByName('Nom').AsString + ' ' + FieldByName('Prenom').AsString;
        ComboBox.AddItem(LNomPrenom, TCBClient.Create(FieldByName('IdClient').AsInteger));
        Next;
      end;
    end;
end;
procedure TDataModule1.ChargementCBVoitures (var ComboBox : TComboBox;
                                             const Requete : String);
(* Charge une combobox avec les voitures correspondant à la requête.
   Chaque élément est constitué d'un texte visible et de données invisibles *)
var
  LVoiture : String; (* Texte visible d'un élément *)
begin
  ComboBox.Clear;
  ChargementVoitures(Requete);
  with SQLQueryVoitures do
    while not EOF do
      begin
        LVoiture := '[' + FieldByName('Plaque').AsString + ']' +
                    FieldByName('Marque').AsString + ' ' +
                    FieldByName('Modele').AsString;
        ComboBox.AddItem(LVoiture, TCBVoiture.Create(FieldByName('Plaque').AsString,
                                                       FieldByName('Cylindree').AsInteger,
                                                       (FieldByName('Transmission').AsString)[1],
                                                       FieldByName('Prix').AsFloat));
        Next;
      end;
    end;
end;
```

## V-F-2 - Requêtes de sélection des voitures et des clients

Ne compilez pas le projet à ce stade, sous peine d'obtenir des erreurs de compilation. Il faut d'abord ajouter l'unité **StdCtrls** à la clause **uses** de l'unité (pour que le compilateur connaisse le type **TComboBox**). Ensuite, les méthodes **ChargementClients** et **ChargementVoitures**, qui avaient été définies plus tôt, doivent être légèrement modifiées pour accepter comme paramètre une autre requête que **SelectionClientsTous** et **SelectionVoituresToutes**.

Nous partons du principe que tous les clients seront chargés dans la liste, et que donc une requête **SELECT \*** toute simple sera suffisante pour remplir le **TSQLQuery** des clients. Par contre, la requête de sélection des voitures devra répondre aux différents filtres de la boîte de dialogue.

Dans le *datamodule*, modifiez la déclaration des méthodes **ChargementClients** et **ChargementVoitures** :

```
procedure ChargementClients (const Requete : String);
procedure ChargementVoitures (const Requete : String);
```

Et leur implémentation :

```
procedure TDataModule1.ChargementClients (const Requete : String);
(* Chargement des clients *)
begin
  with SQLQueryClients do
    begin
      Close;
      SQL.Text := Requete;
```

```

    Open;
  end;
end;
procedure TDataModule1.ChargementVoitures (const Requete : String);
(* Chargement des voitures *)
begin
  with SQLQueryVoitures do
    begin
      Close;
      SQL.Text := Requete;
      Open;
    end;
  end;
end;

```

La modification consiste à passer la requête à exécuter comme paramètre, ce qui rend d'usage beaucoup plus général les deux méthodes.

Cela signifie que nous devons également modifier les appels antérieurs à ces méthodes :

- la méthode [TCarForm.FormShow](#) (de l'unité [Voitures](#)) :

```

procedure TCarForm.FormShow(Sender: TObject);
(* Chargement de la liste des voitures *)
begin
  Enregistre := False;
  DataModule1.ChargementVoitures(SQLSyntax.SelectionVoituresToutes);
end;

```

- la méthode [TCustomerForm.FormShow](#) (si, bien sûr, vous avez fait l'exercice proposé au chapitre V-C-8).

Dans ces deux unités, il faudra rajouter l'unité [SQL](#) dans la clause [uses](#).

Allons maintenant implémenter le chargement des deux *comboboxes* dans notre dialogue [TNewLeasingForm](#). Dans l'éditeur de source, sélectionnez l'onglet [Locations](#) et pressez **F12**. Dans l'inspecteur d'objets, allez dans l'onglet [Événements](#) et cliquez sur les trois points correspondant à l'événement [OnShow](#). Lazarus crée une nouvelle méthode [FormShow](#), que vous complétez comme suit :

```

procedure TNewLeasingForm.FormShow (Sender: TObject);
(* Chargement des contrôles *)
begin
  (* Listes des clients et des voitures *)
  DataModule1.ChargementCBClients(cbClients);
  DataModule1.ChargementCBVoitures(cbVoitures, RequeteSelectionVoituresFiltre);
end;

```

On voit tout de suite qu'il faudra créer la méthode [RequeteSelectionVoituresFiltre](#) ; gardons cela deux minutes dans un coin de notre esprit. Il faut tout d'abord avoir le réflexe de libérer les objets invisibles qui sont liés aux éléments des *comboboxes*, à la fermeture du dialogue.

Cliquez sur les trois points de l'événement [OnClose](#), puis complétez le code :

```

procedure TNewLeasingForm.FormClose(Sender: TObject; var CloseAction: TCloseAction);
(* Détruit les objets passés comme paramètres aux comboboxes *)
var
  Li : Integer;
begin
  for Li := 0 to (cbClients.Items.Count - 1) do
    TCBClient(cbClients.Items.Objects[Li]).Free;
  for Li := 0 to (cbVoitures.Items.Count - 1) do
    TCBVoiture(cbVoitures.Items.Objects[Li]).Free;
  end;
end;

```

Pour compiler, l'unité [DataAccess](#) doit être ajoutée à la clause [uses](#).



Occupons-nous de cette méthode [RequeteSelectionVoituresFiltre](#) dont nous avons postposé la création. Ajoutez cette déclaration dans la section **private** de la classe [TNewLeasingForm](#) :

```
function RequeteSelectionVoituresFiltre : String;
```

Voici son implémentation :

```
function TNewLeasingForm.RequeteSelectionVoituresFiltre: String;
(* Construit la requête de sélection des voitures pour la combobox *)
var
  LTransmission : String;
begin
  if rbTransmissionM.Checked
  then
    LTransmission := 'M'
  else
    if rbTransmissionA.Checked
    then
      LTransmission := 'A'
    else
      LTransmission := '';
  Result := SQLSyntax.SelectionVoituresFiltre(seCylindreeMin.Value, seCylindreeMax.Value,
  LTransmission);
end;
```

Cette méthode ne construit pas directement la requête SQL, elle définit les paramètres à passer à une méthode de la classe [TMySQLSyntax](#), que nous allons tout de suite créer.

Direction l'unité [SQL](#) et notre interface [ISQLSyntax](#). Ajoutez-y la déclaration suivante :

```
function SelectionVoituresFiltre (
  (* Requête de sélection de voitures avec critères *)
  const ACylindreeMin, ACylindreeMax : Integer; (* Cylindrées minimale et maximale *)
  const ATransmission : String (* Transmission : "A" ou "M" *)
) : String;
```

Dans la déclaration du type [TMySQLSyntax](#), ajoutez-la aussi :

```
function SelectionVoituresFiltre (const ACylindreeMin, ACylindreeMax : Integer;
  const ATransmission : String) : String;
```

Un petit **Shift-Ctrl-C** puis complétez :

```
function TMySQLSyntax.SelectionVoituresFiltre (const ACylindreeMin, ACylindreeMax : Integer;
  const ATransmission : String) : String;
(* Requête de sélection de voitures avec critères *)
begin
  Result := 'SELECT * FROM Voitures WHERE Cylindree >= ' + IntToStr(ACylindreeMin) +
    ' AND Cylindree <= ' + IntToStr(ACylindreeMax) + ' ';
  if ATransmission <> ''
  then
    Result := Result + ' AND Transmission = ' + ATransmission + ' ';
  Result := Result + ' ';
end;
```

La requête **SELECT** construite teste les bornes inférieure et supérieure de la cylindrée et introduit un test sur la transmission uniquement si le paramètre [ATransmission](#) n'est pas une chaîne vide.

## V-F-3 - Indices des client et voiture courants

Comme il s'agit d'une nouvelle location, par défaut ce sont les premiers éléments (d'indice 0) des deux *comboboxes* qui sont sélectionnés lorsque s'affiche le dialogue. Cependant, comme la liste des voitures va être rechargée à



chaque fois que les filtres seront modifiés, il faudra garder en mémoire l'indice de la voiture sélectionnée pour la resélectionner après (si elle correspond, bien sûr, toujours aux critères).

Ensuite, rappelez-vous que nous avons prévu de concevoir notre fiche de manière à pouvoir servir de parent à une fiche descendante qui permettrait de modifier une location. Par conséquent, à l'affichage de la fiche, le client et la voiture de la location à modifier devront être sélectionnés. Alors autant prévoir tout de suite de gérer, également, l'indice de la personne.

La conservation des deux indices se fera dans deux propriétés.

Dans la déclaration du type `TNewLeasingForm`, avant la section `private`, créez une section `strict private` et déclarez-y les deux champs suivants :

```
strict private
  FIndexClient : Integer;
  FIndexVoiture : Integer;
```

Dans la section `public`, créez les deux propriétés suivantes :

```
public
  property IndexClient : Integer read FIndexClient write SetIndexClient;
  (* Index du client actuellement sélectionné *)
  property IndexVoiture : Integer read FIndexVoiture write SetIndexVoiture;
  (* Index de la voiture actuellement sélectionnée *)
```

Créez leurs setters dans la section `private` :

```
private
  procedure SetIndexClient (AValue : Integer);
  procedure SetIndexVoiture (AValue : Integer);
```

Voici le code source des setters :

```
procedure TNewLeasingForm.SetIndexClient (AValue : Integer);
(* Setter de la propriété IndexClient *)
begin
  if FIndexClient = AValue
  then
    Exit;
  FIndexClient := AValue;
end;
procedure TNewLeasingForm.SetIndexVoiture (AValue : Integer);
(* Setter de la propriété IndexVoiture *)
begin
  if FIndexVoiture = AValue
  then
    Exit;
  FIndexVoiture := AValue;
end;
```

Initialisons les deux propriétés dans la méthode `FormShow` :

```
procedure TNewLeasingForm.FormShow(Sender: TObject);
(* Chargement des contrôles *)
begin
  (* Listes des clients et des voitures *)
  DataModule1.ChargementCBClients(cbClients);
  DataModule1.ChargementCBVoitures(cbVoitures, RequeteSelectionVoituresFiltre);
  // DÉBUT DE L'AJOUT
  (* Éléments sélectionnés dans les deux comboboxes *)
  IndexClient := 0;
  IndexVoiture := 0;
  cbClients.ItemIndex := IndexClient;
```

```
cbVoitures.ItemIndex := IndexVoiture;
// FIN DE L'AJOUT
end;
```



*Nous savons déjà que, dans la fiche descendante qui permettra de modifier une location, ces deux indices seront initialisés non pas à 0 mais à ceux correspondant au client et à la voiture de la location.*

Nous devons aussi prévoir de mettre à jour les deux propriétés à chaque changement de client ou de voiture. Allez dans l'inspecteur d'objets, sélectionnez la *combobox* **cbClients**, allez dans l'onglet **Événements** et cliquez sur les trois points correspondant à l'événement **OnChange**. Complétez la méthode événementielle créée :

```
procedure TNewLeasingForm.cbClientsChange(Sender: TObject);
(* Mise à jour de l'index de l'élément actuellement sélectionné *)
begin
    IndexClient := cbClients.ItemIndex;
end;
```

Faites de même pour la *combobox* **cbVoitures** :

```
procedure TNewLeasingForm.cbVoituresChange(Sender: TObject);
(* Changement de voiture : mise à jour de l'index *)
begin
    IndexVoiture := cbVoitures.ItemIndex;
end;
```

## V-F-4 - Estimation du prix de la location

Souvenez-vous, nous avons déposé un composant **TStaticText**, nommé **stEstimationPrix**. Nous y afficherons le prix de la location. De quoi dépend ce prix :

- du coût de location journalier (une donnée liée à la voiture) ;
- de la durée (qui dépend des dates de début et de fin, sur la fiche) ;
- de l'assurance (case à cocher sur la fiche).

Créons une méthode **EstimationPrix**, dans la section **private** de la déclaration de type de la fiche :

```
procedure EstimationPrix;
```

Voici son implémentation :

```
procedure TNewLeasingForm.EstimationPrix;
(* Affichage du prix de location à chaque changement de date ou de voiture *)
var
    LPrix : Real;
begin
    if (Length(deDateDebut.Text) > 0) and (Length(deDateFin.Text) > 0) and (cbVoitures.ItemIndex >= 0)
    then
        begin
            LPrix := (DaysBetween(deDateDebut.Date, deDateFin.Date) + 1) *
                TCBVoiture(cbVoitures.Items.Objects[cbVoitures.ItemIndex]).Prix;
            if cbAssurance.Checked
            then
                LPrix := LPrix + 10.00;
            stEstimationPrix.Caption := FloatToStr(LPrix);
        end;
end;
```

La fonction [DaysBetween](#) compte le nombre de jours entre deux dates ; pour pouvoir l'utiliser, vous devez ajouter l'unité [DateUtils](#) à la clause [uses](#). Nous incrémentons son résultat, car par exemple, le client va payer deux jours de location, et non un seul, s'il rend la voiture le lendemain. Notre entreprise de location n'est pas une société philanthropique, non mais !

Regardez où nous allons rechercher le coût journalier dans les données de la voiture : dans l'objet lié à l'élément.

```
TCBVoiture(cbVoitures.Items.Objects[cbVoitures.ItemIndex]).Prix
```

Réfléchissez : à quel moment le prix de location doit-il être calculé ? À l'affichage de la fiche, vous avez raison. Mais encore ? À chaque fois qu'une autre voiture sera sélectionnée, que la durée de la location aura changé et que l'assurance complémentaire sera cochée ou non. Bigre !

Commençons par compléter la méthode [FormShow](#), pour que le prix soit visible dès l'affichage de la fiche :

```
procedure TNewLeasingForm.FormShow(Sender: TObject);
(* Chargement des contrôles *)
begin
    (* Listes des clients et des voitures *)
    DataModule1.ChargementCBClients(cbClients);
    DataModule1.ChargementCBVoitures(cbVoitures, RequeteSelectionVoituresFiltre);
    (* Éléments sélectionnés dans les deux comboboxes *)
    IndexClient := 0;
    IndexVoiture := 0;
    cbClients.ItemIndex := IndexClient;
    cbVoitures.ItemIndex := IndexVoiture;
    // DÉBUT DE L'AJOUT
    (* Estimation du prix *)
    EstimationPrix;
    // FIN DE L'AJOUT
end;
```

Ensuite, complétons la méthode [cbVoituresChange](#) :

```
procedure TNewLeasingForm.cbVoituresChange(Sender: TObject);
(* Changement de voiture : mise à jour de l'index et réestimation du prix de location *)
begin
    IndexVoiture := cbVoitures.ItemIndex;
    // DÉBUT DE L'AJOUT
    EstimationPrix;
    // FIN DE L'AJOUT
end;
```

Réagissons au cochage ou au décochage de la case à cocher de l'assurance complémentaire, en cliquant sur les trois points qui correspondent à l'événement [OnChange](#) du composant [cbAssurance](#), dans l'inspecteur d'objets :

```
procedure TNewLeasingForm.cbAssuranceChange(Sender: TObject);
(* Changement d'option d'assurance : recalcul du prix de location *)
begin
    EstimationPrix;
end;
```

Il reste à réagir au changement de date de début ou de fin de location. Nous ne pouvons nous contenter de recalculer le prix de location, nous devons aussi faire en sorte que la date de fin reste au moins égale à la date de début. Si ce n'est pas le cas, il faudra corriger les dates ; nous ne nous casserons pas la tête : la date de début sera simplement recopiée dans la date de fin.



*Rappelons-nous qu'à l'affichage du dialogue, les deux dates sont initialisées à la date du jour.*

Dans l'inspecteur d'objets, créez une méthode événementielle pour l'événement **OnChange** des deux composants **deDateDebut** et **deDateFin** (voici une des deux, l'autre étant identique) :

```
procedure TNewLeasingForm.deDateDebutChange(Sender: TObject);
(* Changement de la durée : test des dates et réestimation du prix de la location *)
begin
  (* Teste préalablement si la date de fin est postérieure à la date de début *)
  if (Length(deDateDebut.Text) > 0) and (CompareDate(deDateDebut.Date, deDateFin.Date) > 0)
  then (* Erreur : la date de fin devient la date de début *)
    deDateFin.Date := deDateDebut.Date;
  EstimationPrix;
end;
```

## V-F-5 - Réaction aux modifications des filtres

À présent, nous devons prévoir de recharger la liste des voitures à chaque fois que l'utilisateur modifie un des filtres.

Les contrôles dont le changement entraîne la mise à jour de la liste des voitures sont :

- les **TSpinEdit** **seCylindreeMin** et **seCylindreeMax** ;
- les cases à cocher contenues dans le groupe de boutons radio **rgTransmission**.

De façon identique pour les deux premiers contrôles cités, dans l'inspecteur d'objets, allez dans l'onglet **Événements** et cliquez sur les trois points à côté de l'événement **OnChange**, puis complétez la méthode événementielle comme suit :

```
procedure TNewLeasingForm.seCylindreeMinChange(Sender: TObject);
(* Modification du filtre sur la cylindrée : rechargement des voitures *)
begin
  DataModule1.ChargementCBVoitures(cbVoitures, RequeteSelectionVoituresFiltre);
  cbVoitures.ItemIndex := IndexVoiture;
end;
```

Pour le groupe de boutons radio, c'est en regard de l'événement **OnChange** de chaque bouton qu'il faut cliquer. Le code de la méthode événementielle est le suivant (par exemple, pour le choix de la transmission automatique (« A »)) :

```
procedure TNewLeasingForm.rbTransmissionAChange(Sender: TObject);
(* Modification du filtre sur la transmission : rechargement des voitures *)
begin
  DataModule1.ChargementCBVoitures(cbVoitures, RequeteSelectionVoituresFiltre);
  cbVoitures.ItemIndex := IndexVoiture;
end;
```

## V-F-6 - Test de disponibilité de la voiture

Avant d'enregistrer une nouvelle location, il faut évidemment s'assurer que la voiture est bien disponible pendant toute la durée de la période souhaitée ! Comment faire ? En créant une requête qui compte toutes les locations de ladite voiture entre les dates de début et de fin. Direction l'unité **SQL** et l'interface **ISQLSyntax**, dans laquelle vous ajoutez la méthode suivante :

```
function SelectionVoitureLouee (
  (* Requête de sélection de location correspondant aux critères *)
  const APlaque : String; (* Plaque de la voiture *)
  const ADateDebut, ADateFin : TDateTime (* Dates de début et de fin de location *)
) : String;
```

Cette méthode doit obligatoirement figurer dans la déclaration de la classe **TMySQLSyntax**, et voici son implémentation :

```
function TMySQLSyntax.SelectionVoitureLouee (const APlaque : String;
                                             const ADateDebut, ADateFin : TDateTime) : String;
(* Requête de sélection de location correspondant aux critères *)
begin
    Result := 'SELECT * FROM Locations' +
              ' INNER JOIN Voitures ON Locations.Plaque = Voitures.Plaque' +
              ' INNER JOIN Clients ON Locations.IdClient = Clients.IdClient' +
              ' WHERE Locations.Plaque = ''' + APlaque + ''' +
              ' AND (DateDebut <= ''' + DateToStr(ADateDebut, FormatDate) + ''')' +
              ' AND (DateFin >= ''' + DateToStr(ADateFin, FormatDate) + ''')';
end;
```

Rien de très spécial dans cette requête, dans laquelle vous remarquez la jointure des tables [Voitures](#) et [Clients](#) et l'utilisation de la structure [FormatDate](#), qui est, rappelons-le, également déclarée dans l'interface pour définir le format de date propre au système de bases de données.

La nouvelle requête va être utilisée par une fonction qui va compter le nombre d'enregistrements correspondant à la voiture et aux dates de location. Cette fonction est logiquement définie dans le *datamodule*, et donc prenez la direction de l'unité [DataAccess](#).

Dans la déclaration du *datamodule* [TDataModule1](#), ajoutez cette fonction :

```
function VoitureDejaLouee (const APlaque : String;
                           const ADateDebut, ADateFin : TDateTime) : Boolean;
```

Prenons le temps de réfléchir : la requête peut-elle être exécutée par le [TSQLQuery](#) correspondant à la table [Locations](#), [TSQLQueryMain](#) ? Si nous faisons cela, le [TDBGrid](#) de la fenêtre principale, qui contient la liste des locations et qui est lié au [SQLQueryMain](#), va automatiquement être rechargé avec les locations qui répondent à la requête de comptage que nous exécutons. Ce n'est pas ce que nous voulons, ce [TDBGrid](#) doit rester inchangé. Voici une autre solution : ajouter un nouveau [TSQLQuery](#) au *datamodule*, que nous appellerons [SQLQueryTemp](#) (pour « temporaire »). Je vous laisse faire cela tout(e) seule, vous le faites les yeux fermés à présent !



*Il n'y a pas besoin de composant [TDataSource](#), puisque nous ne travaillons qu'avec des contrôles classiques (seuls les composants spécialisés bases de données le requièrent).*

Vous avez terminé ? Voici l'implémentation de la fonction [VoitureDejaLouee](#) :

```
function TDataModule1.VoitureDejaLouee (const APlaque : String;
                                         const ADateDebut, ADateFin : TDateTime) : Boolean;
(* Effectue une requête pour voir si une voiture est louée entre deux dates.
   Le principe est de rechercher les locations de ladite voiture pendant ce laps de temps *)
begin
    SQLQueryTemp.Close;
    SQLQueryTemp.SQL.Text := SQLSyntax.SelectionVoitureLouee(APlaque, ADateFin, ADateDebut);
    SQLQueryTemp.Open;
    Result := SQLQueryTemp.RecordCount > 0;
end;
```

Le principe est d'appliquer la requête de sélection et de tester le nombre d'enregistrements correspondants (donc, de compter combien de fois la voiture est louée, totalement ou partiellement, entre les dates de début et de fin).

## V-F-7 - Enregistrement de la location

Exactement comme nous l'avions fait dans la fiche [TCarForm](#) (qui permettait de modifier la table des voitures) - et dans la fiche [TCustomerForm](#) si vous aviez fait l'exercice proposé - nous créons un champ [FEnregistre](#) dans la section [strict private](#), une propriété [Enregistre](#) dans la section [public](#) et un setter dans la section [private](#) :

```
strict private
```

```
// DÉBUT DE L'AJOUT
FEnregistre : Boolean;
// FIN DE L'AJOUT
FIndexClient : Integer;
FIndexVoiture : Integer;
private
// DÉBUT DE L'AJOUT
procedure SetEnregistre (AValue : Boolean);
// FIN DE L'AJOUT
procedure SetIndexClient (AValue : Integer);
procedure SetIndexVoiture (AValue : Integer);
```

```
public
// DÉBUT DE L'AJOUT
property Enregistre : Boolean read FEnregistre write SetEnregistre;
(* Indique si les données ont été enregistrées *)
// FIN DE L'AJOUT
property IndexClient : Integer read FIndexClient write FIndexClient;
(* Index du client actuellement sélectionné *)
property IndexVoiture : Integer read FIndexVoiture write FIndexVoiture;
(* Index de la voiture actuellement sélectionnée *)
```

Cette propriété doit être initialisée dès l'affichage de la fiche, dans la méthode [FormShow](#) :

```
procedure TNewLeasingForm.FormShow(Sender: TObject);
(* Chargement des contrôles *)
begin
// DÉBUT DE L'AJOUT
Enregistre := False;
// FIN DE L'AJOUT
(* Listes des clients et des voitures *)
// Etc.
end;
```

N'oublions pas le code du setter :

```
procedure TNewLeasingForm.SetEnregistre (AValue : Boolean);
(* Setter de la propriété Enregistre *)
begin
if FEnregistre = AValue
then
Exit;
FEnregistre := AValue;
end;
```

Je vous laisse vous occuper de la création des méthodes correspondant aux deux boutons « Enregistrer » et « Annuler ». Juste leur création, car nous nous pencherons attentivement sur leur contenu.

Créez également une méthode [FormCloseQuery](#), qui sera de la même forme que précédemment :

```
procedure TNewLeasingForm.FormCloseQuery(Sender: TObject; var CanClose: boolean);
(* Message d'avertissement en cas de fermeture sans sauvegarde *)
begin
if Enregistre
then
CanClose := True
else
CanClose := (MessageDlg('Voulez-vous fermer sans enregistrer ?', mtConfirmation, [mbYes,
mbNo], 0) = mrYes);
end;
```

Répondons à présent au clic sur le bouton « Enregistrer ».

Premièrement, avant d'enregistrer nous devons vérifier que la location est possible, grâce à la méthode [VoitureDejaLouee](#) que nous avons créée dans le *datamodule* il y a quelques minutes :

```
procedure TUpdateLeasingForm.btnEnregistrerClick(Sender: TObject);
(* Sauvegarde de la location *)
begin
    (* Requête vérifiant que la voiture n'est pas déjà louée aux dates sélectionnées *)
    if DataModule1.VoitureDejaLouee(LocationAModifier.IdLocation,
    TCBVoiture(cbVoitures.Items.Objects[cbVoitures.ItemIndex]).Plaque, deDateDebut.Date,
    deDateFin.Date)
    then
        MessageDlg('La voiture est déjà louée pendant cette période', mtError, [mbOk], 0)
    else
        begin
            // Enregistrement
        end;
end;
```

L'enregistrement proprement dit de la location va se faire dans le *datamodule*. Ajoutez-y la méthode suivante, en dessous de celles destinées à la sauvegarde des voitures et des clients :

```
function SauvegardeLocations (const Requete : String) : Boolean;
```

Voici son implémentation :

```
function TDataModule1.SauvegardeLocations (const Requete : String) : Boolean;
(* Sauvegarde de la table Locations *)
begin
    SQLQueryMain.Close;
    SQLQueryMain.SQL.Clear;
    SQLQueryMain.SQL.Add(Requete);
    SQLQueryMain.ExecSQL;
    Result := Commit;
end;
```

Il n'y a plus qu'à construire la requête qui est passée comme paramètre, dans l'unité [SQL](#). Il s'agit d'une requête d'insertion :

```
function InsertionLocation (
    (* Requête d'insertion d'une nouvelle location *)
    const AIdClient : Integer;           (* Identificateur du client *)
    const APlaque : String;             (* Plaque de la voiture *)
    const ADateDebut, ADateFin : TDateTime; (* Dates de début et de fin *)
    const AAssurance: Boolean          (* Option d'assurance complémentaire *)
) : String;
```

Dont voici l'implémentation :

```
function TMySQLSyntax.InsertionLocation (const AIdClient : Integer;
                                         const APlaque : String;
                                         const ADateDebut, ADateFin : TDateTime;
                                         const AAssurance : Boolean) : String;
(* Requête d'insertion d'une nouvelle location *)
begin
    Result := 'INSERT INTO Locations VALUES (NULL, ''' + IntToStr(AIdClient) + ''', ''' + APlaque
    + ''', ''' +
        DateToStr(ADateDebut, FormatDate) + ''', ''' +
        DateToStr(ADateFin, FormatDate) + ''', NULL, ' ';
    if AAssurance
    then
        Result := Result + '1);'
    else
        Result := Result + '0);';
end;
```

Vous constatez que le client et la voiture sont ajoutés sous la forme des deux clés étrangères [IdClient](#) et [Plaque](#).

Nous devons encore compléter la méthode [TNewLeasingForm.btnEnregistrerClick](#) :

```
procedure TNewLeasingForm.btnEnregistrerClick(Sender: TObject);
(* Sauvegarde de la location *)
begin
    (* Requête vérifiant que la voiture n'est pas déjà louée aux dates sélectionnées *)
    if
        DataModule1.VoitureDejaLouee(TCBVoiture(cbVoitures.Items.Objects[cbVoitures.ItemIndex]).Plaque,
        deDateDebut.Date, deDateFin.Date)
    then
        MessageDlg('La voiture est déjà louée pendant cette période', mtError, [mbOk], 0)
    else
        begin
            // DÉBUT DE L'AJOUT
            Enregistre := DataModule1.SauvegardeLocations(SQLSyntax.InsertionLocation(
                TCBCClient(cbClients.Items.Objects[cbClients.ItemIndex]).IdClient,
                TCBVoiture(cbVoitures.Items.Objects[cbVoitures.ItemIndex]).Plaque,
                deDateDebut.Date, deDateFin.Date, cbAssurance.Checked));
            Close;
            // FIN DE L'AJOUT
        end;
end;
```

Une dernière formalité, la réponse au clic sur le bouton « Annuler » :

```
procedure TNewLeasingForm.btnAnnulerClick(Sender: TObject);
(* Fermeture du dialogue *)
begin
    Close;
end;
```



**Testez la création de nouvelles locations, mais n'oubliez pas qu'elles seront réellement insérées dans la base de données !**

## V-F-8 - Une classe descendante pour le dialogue de modification de location

En fait, pour modifier une location existante, il n'y a pas besoin de créer de toutes pièces un nouveau dialogue : quelques modifications au dialogue de création de location suffiront. Nous allons donc créer une classe descendante et redéfinir ou ajouter l'une ou l'autre méthode ou propriété.

Faisons le point sur ce que ce dialogue descendant aura de différent par rapport à celui d'origine :

- au démarrage, il faudra sélectionner le bon client et la bonne voiture, initialiser les dates de la location et l'option d'assurance ;
- à l'enregistrement, lors de la recherche des locations en cours, il faudra évidemment exclure la location que nous sommes en train de modifier (sinon jamais nous ne pourrions l'enregistrer).

Nous devons donc redéfinir les méthodes [FormShow](#) et [btnEnregistrerClick](#).

Allez-y : dans la section [type](#) de l'interface de l'unité [Locations](#), créez la classe descendante [TUpdateLeasingForm](#) :

```
TUpdateLeasingForm = class(TNewLeasingForm)
    procedure FormShow(Sender: TObject);
    procedure btnEnregistrerClick(Sender: TObject);
end;
```



## V-F-8-a - Initialisation des champs

Comment allons-nous transmettre au dialogue les données de la location à modifier ? Un moyen parmi d'autres est sous forme d'un objet similaire à ceux qui servent à stocker les données des voitures et des clients dans les *comboboxes*, [TCBVoiture](#) et [TCBClient](#).

Direction l'unité [DataAccess](#), créez une nouvelle classe [TLocation](#) en dessous des deux classes qui viennent d'être citées :

```
TLocation = class
  (* Données initiales d'une location à modifier *)
  strict private
    FIdLocation : Integer;
    FPlaque : String;
    FIdClient : Integer;
    FDateDebut : TDateTime;
    FDateFin : TDateTime;
    FAssurance : Boolean;
  public
    property IdLocation : Integer read FIdLocation;
    property Plaque : String read FPlaque;
    property IdClient : Integer read FIdClient;
    property DateDebut : TDateTime read FDateDebut;
    property DateFin : TDateTime read FDateFin;
    property Assurance : Boolean read FAssurance;
    constructor Create (const AIdLocation : Integer;
                       const APlaque : String;
                       const AIdClient : Integer;
                       const ADateDebut, ADateFin : TDateTime;
                       const AAssurance : Boolean);
end;
```

Voici le contenu du constructeur [Create](#) :

```
constructor TLocation.Create (const AIdLocation : Integer;
                             const APlaque : String;
                             const AIdClient : Integer;
                             const ADateDebut, ADateFin : TDateTime;
                             const AAssurance : Boolean);

(* Initialisation des champs *)
begin
  FIdLocation := AIdLocation;
  FPlaque := APlaque;
  FIdClient := AIdClient;
  FDateDebut := ADateDebut;
  FDateFin := ADateFin;
  FAssurance := AAssurance;
end;
```

Lors de l'appel du dialogue de modification, une structure de type [TLocation](#), contenant les données de l'application à modifier, sera transmise au dialogue, dans une propriété que nous allons tout de suite créer.

Dans la déclaration du type [TUpdateLeasingForm](#), ajoutez ce qui suit dans les sections [strict private](#) et [public](#) :

```
TUpdateLeasingForm = class(TNewLeasingForm)
  procedure FormShow(Sender: TObject);
  procedure btnEnregistrerClick(Sender: TObject);
  // DÉBUT DE L'AJOUT
  strict private
    FLocationAModifier : TLocation;
  private
    procedure SetLocationAModifier (AValue : TLocation);
  public
    property LocationAModifier : TLocation read FLocationAModifier write SetLocationAModifier;
    (* Données pour initialiser les contrôles *)
end;
```

```
// FIN DE L'AJOUT  
end;
```

Pressez **Shift-Ctrl-C** pour créer le setter et les deux méthodes dans la section [implementation](#). Commençons par le classique setter :

```
procedure TUpdateLeasingForm.SetLocationAModifier (AValue : TLocation);  
(* Setter de la location à modifier *)  
begin  
    if FLocationAModifier = AValue  
    then  
        Exit;  
    FLocationAModifier := AValue;  
end;
```

Penchons-nous ensuite sur [FormShow](#).

Tout d'abord, le chargement des *comboboxes* peut être hérité de la classe parent. La méthode [FormShow](#) commencera ainsi :

```
inherited FormShow(Sender);
```

C'est après que les *comboboxes* auront été chargées que nous allons y sélectionner le bon client et la bonne voiture, initialiser les dates de location et cocher l'assurance complémentaire s'il y a lieu. Tout cela se trouve, rappelons-le, dans la propriété [LocationAModifier](#), qui aura été initialisée par la fenêtre parent. Voici le code de la méthode complète :

```
procedure TUpdateLeasingForm.FormShow(Sender: TObject);  
(* Initialisation des contrôles *)  
var  
    Li : Integer;          (* Indice dans une combobox *)  
    LTrouve : Boolean;      (* Voiture ou client à modifier trouvé dans sa combobox *)  
begin  
    (* Initialisation par défaut des contrôles *)  
    inherited FormShow(Sender);  
    (* Adaptation des contrôles aux données à modifier *)  
    deDateDebut.Date := LocationAModifier.DateDebut;  
    deDateFin.Date := LocationAModifier.DateFin;  
    if LocationAModifier.Assurance  
    then  
        cbAssurance.Checked := True;  
    LTrouve := False;  
    Li := 0;  
    while (Li < cbClients.Items.Count) and not LTrouve do  
        if TCBClient(cbClients.Items.Objects[Li]).IdClient = LocationAModifier.IdClient  
        then  
            begin  
                cbClients.ItemIndex := Li;  
                LTrouve := True;  
            end  
        else  
            Inc(Li);  
        LTrouve := False;  
        Li := 0;  
        while (Li < cbVoitures.Items.Count) and not LTrouve do  
            if TCBVoiture(cbVoitures.Items.Objects[Li]).Plaque = LocationAModifier.Plaque  
            then  
                begin  
                    cbVoitures.ItemIndex := Li;  
                    LTrouve := True;  
                end  
            else  
                Inc(Li);  
            end;  
        end;  
    end;
```

Dans l'unité [Main](#), dans la fiche principale de l'application, voyons tout de suite le code d'exécution du dialogue [TUpdateLeasingForm](#), et spécialement comment l'objet de type [TLocation](#) est initialisé et transmis. Cela se fera dans la méthode qui répondra à l'événement [OnClick](#) du bouton [btnModifier](#) :

```
procedure TMainForm.btnModifierClick (Sender : TObject);
(* Modification de la location actuellement sélectionnée *)
var
  LUpdateLeasingForm : TUpdateLeasingForm; (* Dialogue de modification *)
  LLocationAModifier : TLocation; (* Données à modifier *)
begin
  LUpdateLeasingForm := TUpdateLeasingForm.Create(Self);
  try
    (* Récolte des données de l'élément à modifier *)
    with DataModule1.SQLQueryMain do
      LLocationAModifier := TLocation.Create(FieldByName('IdLocation').AsInteger,
                                             FieldByName('Plaque').AsString,
                                             FieldByName('IdClient').AsInteger,
                                             FieldByName('DateDebut').AsDateTime,
                                             FieldByName('DateFin').AsDateTime,
                                             FieldByName('Assurance').AsBoolean);

      LUpdateLeasingForm.LocationAModifier := LLocationAModifier;
      (* Exécution du dialogue *)
      LUpdateLeasingForm.ShowModal;
    finally
      FreeAndNil(LUpdateLeasingForm);
      LLocationAModifier.Free;
    end;
    (* Mise à jour de la liste des locations affichée *)
    DataModule1.ChargementLocations(SQLSyntax.SelectionLocationsFiltre(deFiltreDateDebut.Date,
                                                                    deFiltreDateFin.Date,
                                                                    cbFiltreAssurance.Checked,
                                                                    cbFiltreEnCours.Checked));
  end;
```

Vous voyez que l'initialisation des données de la location à modifier est réalisée avant l'affichage du dialogue (avant [ShowModal](#)). Les données sont récupérées dans l'enregistrement actuellement sélectionné dans le [SQLQuery](#) principal ; les champs sont identifiés à l'aide de la fonction [FieldByName](#) et de leur identificateur dans la base de données.

L'objet créé est assigné à la propriété [LocationAModifier](#) du dialogue, comme annoncé. Après la fermeture du dialogue, nous n'oublions pas de le détruire.

Notez également que nous n'avons pas oublié de mettre à jour la liste des locations en cours dans la fiche principale, après l'exécution du dialogue.

## V-F-8-b - Enregistrement de la location modifiée

Il ne nous reste qu'à apporter des changements à l'enregistrement de la location. Comme nous l'avons dit auparavant, si nous n'excluons pas la location courante de la recherche des locations en cours, jamais nous ne pourrions enregistrer les changements : le message disant que la voiture est déjà louée apparaîtra systématiquement. Pas convaincu(e) ? Faites l'expérience de ne pas redéfinir la méthode [btnEnregistrerClick](#) et exécutez le programme. Impossible d'enregistrer !

Il nous faut donc compléter la requête SQL qui sélectionne toutes les locations de la voiture entre les dates de début et de fin. Pour l'instant, cette requête est construite ainsi (dans l'unité [SQL](#)) :

```
function TMySQLSyntax.SelectionVoitureLouee (const APlaque : String;
                                             const ADateDebut, ADateFin : TDateTime) : String;
(* Requête de sélection de location correspondant aux critères *)
begin
  Result := 'SELECT * FROM Locations' +
            ' INNER JOIN Voitures ON Locations.Plaque = Voitures.Plaque' +
            ' INNER JOIN Clients ON Locations.IdClient = Clients.IdClient' +
```

```
' WHERE Locations.Plaque = ''' + APlaques + ''' +
' AND (DateDebut <= ''' + DateToStr(ADateDebut, FormatDate) + ''')' +
' AND (DateFin >= ''' + DateToStr(ADateFin, FormatDate) + ''')';
end;
```

Il faut ajouter une condition qui teste si le champ **IdLocation** est différent de celui de la location qui est en train d'être modifiée.

Dans l'interface **ISQLSyntax** et dans la classe **TMySQLSyntax**, créez une nouvelle version de la méthode **SelectionVoitureLouee** :

```
function SelectionVoitureLouee (
  (* Requête de sélection de location correspondant aux critères, excluant la location
  courante *)
  const AIdLocationAExclure : Integer;      (* Identificateur de la location à exclure *)
  const APlaques : String;                  (* Plaque de la voiture *)
  const ADateDebut, ADateFin : TDateTime    (* Dates de début et de fin de location *)
) : String;
```

Son implémentation est presque identique à l'ancienne version, on y ajoute la nouvelle condition :

```
function TMySQLSyntax.SelectionVoitureLouee (const AIdLocationAExclure : Integer;
  const APlaques : String;
  const ADateDebut, ADateFin : TDateTime) : String;
(* Requête de sélection de location correspondant aux critères, excluant la location courante *)
begin
  Result := 'SELECT * FROM Locations' +
  ' INNER JOIN Voitures ON Locations.Plaque = Voitures.Plaque' +
  ' INNER JOIN Clients ON Locations.IdClient = Clients.IdClient' +
  // DÉBUT DE LA NOUVELLE CONDITION
  ' WHERE Locations.IdLocation <= ''' + IntToStr(AIdLocationAExclure) + ''' +
  // FIN DE LA NOUVELLE CONDITION
  ' AND Locations.Plaque = ''' + APlaques + ''' +
  ' AND (DateDebut <= ''' + DateToStr(ADateDebut, FormatDate) + ''')' +
  ' AND (DateFin >= ''' + DateToStr(ADateFin, FormatDate) + ''')';
end;
```



On utilise l'opérateur « != » issu du C, qui est plus standard que l'opérateur « <> ».

Grâce au nombre différent de paramètres, le compilateur ne risque pas de se tromper entre les deux versions de la méthode, même si elles ont le même nom.

Même chose dans le *datamodule*, puisque nous allons créer une nouvelle version de la méthode **VoitureDejaLouee** :

```
function VoitureDejaLouee (const AIdLocationAExclure : Integer;
  const APlaques : String;
  const ADateDebut, ADateFin : TDateTime) : Boolean;
```

Dont voici le code source :

```
function TDataModule1.VoitureDejaLouee (const AIdLocationAExclure : Integer;
  const APlaques : String;
  const ADateDebut, ADateFin : TDateTime) : Boolean;
(* Effectue une requête pour voir si une voiture est louée entre deux dates.
  Le principe est de rechercher les locations de ladite voiture pendant ce laps de temps.
  Version excluant la location en cours de modification *)
begin
  SQLQueryTemp.Close;
  SQLQueryTemp.SQL.Text := SQLSyntax.SelectionVoitureLouee(AIdLocationAExclure, APlaques,
  ADateFin, ADateDebut);
  SQLQueryTemp.Open;
  Result := SQLQueryTemp.RecordCount > 0;
```

```
end;
```

Il nous reste une ultime requête SQL à créer. Nous avons déjà utilisé les commandes **SELECT** (pour sélectionner des enregistrements) et **INSERT** (pour en ajouter de nouveaux) ; pour la mise à jour d'un enregistrement, nous avons besoin de la requête **UPDATE**.

Dans l'interface **ISQLSyntax** et dans la classe **TMySQLSyntax** de l'unité **SQL**, ajoutez cette méthode :

```
function ModificationLocation (
    (* Requête de modification d'une location *)
    const AIdLocation : Integer;          (* Identificateur de la location modifiée *)
    const AIdClient : Integer;            (* Identificateur du client *)
    const APlaque : String;               (* Plaque de la voiture *)
    const ADateDebut, ADateFin : TDateTime; (* Dates de début et de fin *)
    const AAssurance: Boolean             (* Option d'assurance complémentaire *)
) : String;
```

Voici son implémentation :

```
function TMySQLSyntax.ModificationLocation (const AIdLocation, AIdClient : Integer;
                                           const APlaque : String;
                                           const ADateDebut, ADateFin : TDateTime;
                                           const AAssurance : Boolean) : String;

(* Requête de modification d'une location *)
begin
    Result := 'UPDATE Locations SET Plaque = ''' + APlaque + ''', IdClient = ''' +
    IntToStr(AIdClient) +
    ''', DateDebut = ''' + DateToStr(ADateDebut, FormatDate) +
    ''', DateFin = ''' + DateToStr(ADateFin, FormatDate) + ''', Assurance = ''';

    if AAssurance
    then
        Result := Result + '1'''
    else
        Result := Result + '0''';
    Result := Result + ' WHERE IdLocation = ''' + IntToStr(AIdLocation) + ''';'
end;
```

Nous avons presque terminé. Voici notre méthode **TUpdateLeasingForm.btnEnregistrerClick** :

```
procedure TUpdateLeasingForm.btnEnregistrerClick(Sender: TObject);
(* Sauvegarde de la location *)
begin
    (* Requête vérifiant que la voiture n'est pas déjà louée aux dates sélectionnées *)
    if DataModule1.VoitureDejaLouee(LocationAModifier.IdLocation,
    TCBVoiture(cbVoitures.Items.Objects[cbVoitures.ItemIndex]).Plaque, deDateDebut.Date,
    deDateFin.Date)
    then
        MessageDlg('La voiture est déjà louée pendant cette période', mtError, [mbOk], 0)
    else
        begin
            Enregistre := DataModule1.SauvegardeLocations(SQLSyntax.ModificationLocation(
                LocationAModifier.IdLocation,
                TCBCClient(cbClients.Items.Objects[cbClients.ItemIndex]).IdClient,
                TCBVoiture(cbVoitures.Items.Objects[cbVoitures.ItemIndex]).Plaque,
                deDateDebut.Date, deDateFin.Date, cbAssurance.Checked));

            Close;
        end;
end;
```

L'identificateur **IdLocation** est issu de la structure **LocationAModifier**.

## V-F-9 - Retour d'une voiture louée

Lorsqu'une voiture louée est restituée, il faut mettre à jour la base de données en ajoutant une date de rentrée à la location.

Sur la fiche principale, ajoutez un bouton « &Rentrée de la voiture », nommé `btnRentreeVoiture`, en dessous du bouton « Supprimer la location ». Dans l'inspecteur d'objets, cliquez en regard de l'événement `OnClick` de ce nouveau bouton, pour créer une procédure événementielle que nous compléterons dans quelques minutes.



*Pour ne pas compliquer l'application, nous considérons que la date de rentrée de la voiture est la date du jour. Mais vous pouvez, comme exercice, utiliser un dialogue tel que `TCalendarDialog` pour permettre à l'utilisateur de choisir une date.*

Comme il s'agit de la mise à jour d'un enregistrement, la commande SQL que nous devons utiliser est **UPDATE**. Dans l'unité SQL, créez cette nouvelle méthode :

```
function ModificationLocation (
    (* Requête de modification de la date de rentrée d'une voiture *)
    const AIdLocation : Integer;      (* Identificateur de la location modifiée *)
    const ADateRentree : TDateTime    (* Dates de début et de fin *)
) : String;
```

Elle se différencie de la méthode de même nom existante par ses paramètres. Voici son contenu :

```
function TMySQLSyntax.ModificationLocation (const AIdLocation : Integer;
                                             const ADateRentree : TDateTime) : String;
(* Requête de modification de la date de rentrée d'une voiture *)
begin
    Result := 'UPDATE Locations SET DateRentree = ''' + DateToStr(ADateRentree, FormatDate) +
              ''' WHERE IdLocation = ''' + IntToStr(AIdLocation) + ''';
end;
```

Complétons la méthode événementielle `TMainForm.btnRentreeVoitureClick`, dans l'unité `Main` :

```
procedure TMainForm.btnRentreeVoitureClick (Sender : TObject);
(* Ajout de la date de rentrée de la voiture *)
begin
    DataModule1.SauvegardeLocations(SQLSyntax.ModificationLocation(DataModule1.SQLQueryMain.FieldByName('IdLocation'
                                                                                                     Today));
    (* Mise à jour de la liste des locations affichée *)
    DataModule1.ChargementLocations(SQLSyntax.SelectionLocationsFiltre(deFiltreDateDebut.Date,
                                                                        deFiltreDateFin.Date,
                                                                        cbFiltreAssurance.Checked,
                                                                        cbFiltreEnCours.Checked));
end;
```

La date du jour est déterminée par la fonction `Today`, de l'unité `DateUtils`.

## V-F-10 - Exercice : supprimer une location

Encore un exercice ?!? Oui, vous devriez être capable de gérer la suppression de la location actuellement sélectionnée dans le DBGrid principal (en réponse à un clic sur le bouton `btnSupprimer`). Une proposition de solution se trouve dans le code source du projet.

Je vous explique quand même quelle commande SQL vous devez utiliser : **DELETE**. Votre requête devra ressembler à ceci :

```
DELETE FROM Locations WHERE IdLocation = '1234';
```

Passez votre requête comme paramètre à la méthode [TDataModule1.SauvegardeLocations](#).

Si vous ne savez pas comment retrouver l'identificateur de la location sélectionnée, voici un indice :

```
DataModule1.SQLQueryMain.FieldName('IdLocation').AsInteger
```

## V-G - Toujours plus loin : une facture avec LazReport

Attention, n'attendez pas de ce chapitre beaucoup d'explications sur la création d'états avec LazReport. Le but du jeu est de montrer l'interaction des composants avec la base de données. Si vous avez besoin d'explications pour démarrer avec LazReport, je vous conseille ce tutoriel très clair sur la [création d'un état simple](#), écrit par Jean-Paul Humbert.

Avant de commencer, prenons le temps de réfléchir. Le but est de créer une facture ; il s'agira d'une facture sur une seule location à la fois. Si nous lions les composants de LazReport au SQLQuery de la fiche principale, cela va créer des factures pour toutes les locations présentes dans le DBGrid ; la solution est de sélectionner une seule facture dans le SQLQuery temporaire, et donc de lier les composants de LazReport au [SQLQueryTemp](#).

### V-G-1 - Requête de sélection

Zou, première étape : créer une requête de sélection de la location pour laquelle nous désirons une facture. Dans l'unité [SQL](#), ajoutez la méthode suivante à l'interface [ISQLSyntax](#) et à la classe [TMySQLSyntax](#) :

```
function SelectionLocationsFiltre (
  (* Requête de sélection d'une location à partir de son identificateur *)
  const AIdLocation : Integer (* Identificateur de la location *)
) : String;
```

Son implémentation est relativement simple :

```
function TMySQLSyntax.SelectionLocationsFiltre (const AIdLocation : Integer) : String;
(* Requête de sélection d'une location à partir de son identificateur *)
begin
  Result := 'SELECT * FROM Locations' +
    ' INNER JOIN Voitures ON Locations.Plaque = Voitures.Plaque' +
    ' INNER JOIN Clients ON Locations.IdClient = Clients.IdClient' +
    ' WHERE IdLocation = ''' + IntToStr(AIdLocation) + ''';';
end;
```

### V-G-2 - Composants LazReport

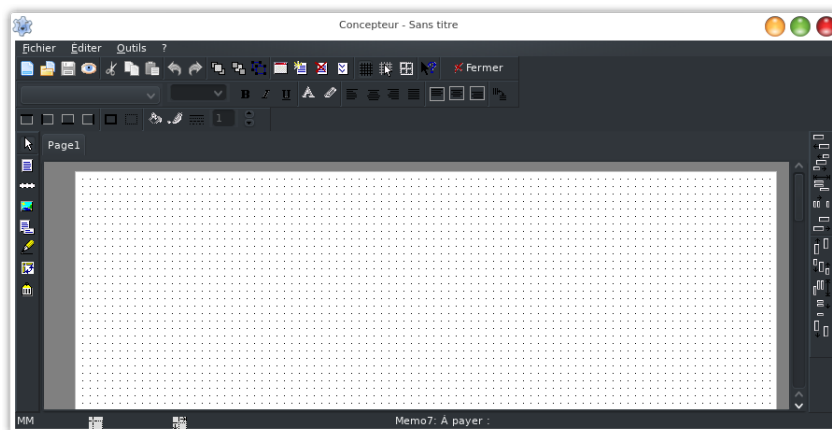
Seconde étape : ajouter les composants LazReport. À quel endroit, d'après vous ? Oui, dans le *datamodule*, avec les autres composants de connexion à la base de données. Direction l'unité [DataAccess](#), pressez **F12** pour afficher le concepteur.

Depuis l'onglet [Data Access](#), déposez un composant [TDataSource](#), que vous renommez [DataSourceTemp](#) et que vous liez à [SQLQueryTemp](#). Déposez ensuite, depuis l'onglet [LazReport](#) :

- un composant [TfrDBDataSet](#) (le second sur l'onglet de la palette), dont vous liez la propriété [DataSet](#) au [SQLQueryTemp](#) ;
- un composant [TfrReport](#) (le tout premier), dont vous liez la propriété [DataSet](#) au [frDBDataSet1](#) que vous venez de déposer.

## V-G-3 - Conception du rapport

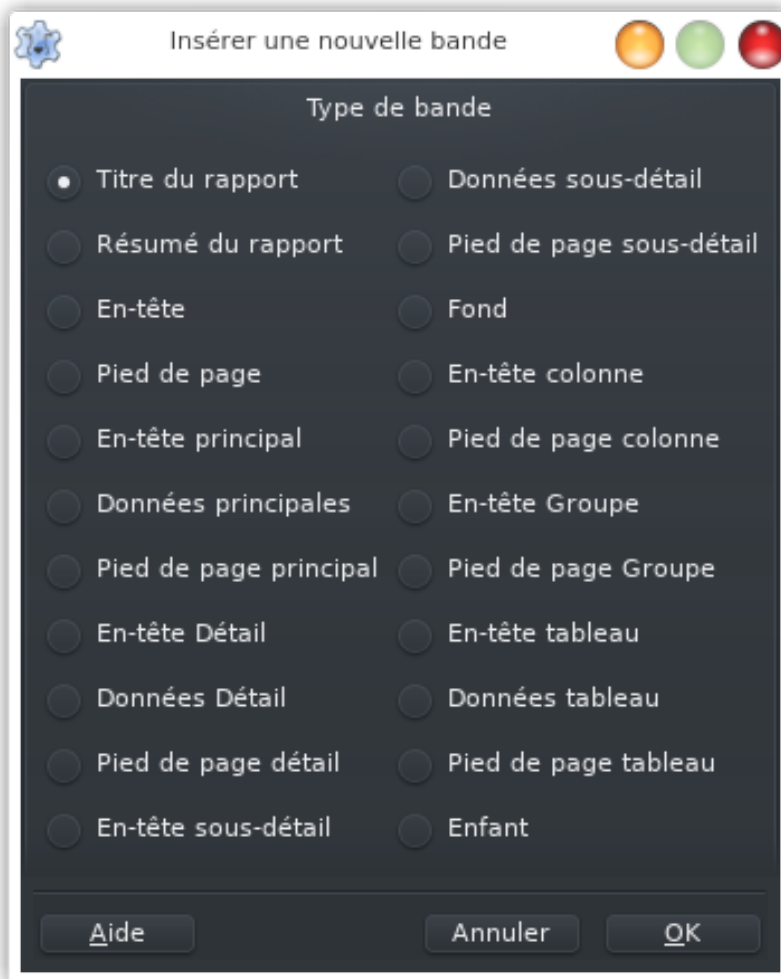
Il s'agit à présent de faire la maquette de la facture. Faites un double-clic sur le composant `frReport1`, une nouvelle fenêtre s'affiche :



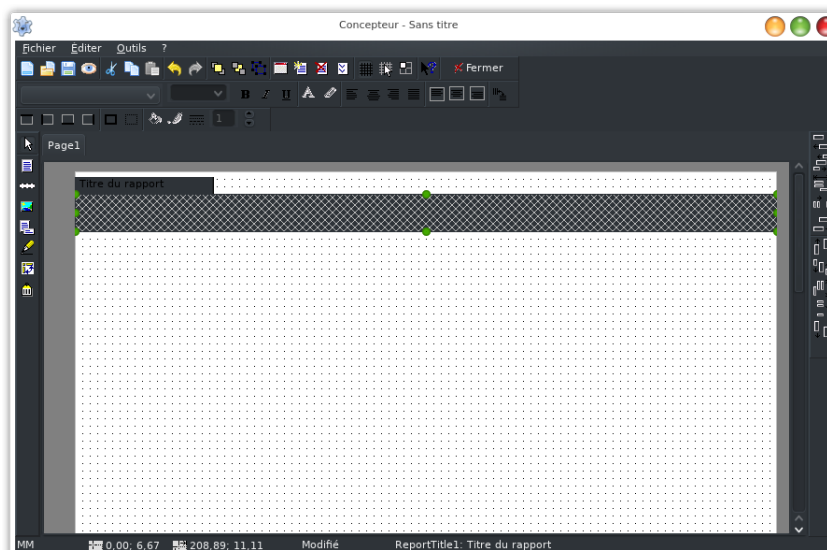
C'est le concepteur de rapport de LazReport.

Créons une en-tête pour la facture, en cliquant sur le tout petit rectangle entouré de pointillés dans la colonne de gauche. Cliquez ensuite en haut et à gauche de la page vide : un rectangle pointillé se dépose et LazReport vous demande de quel type de bande il s'agit. Comme proposé automatiquement, il s'agit du « Titre du rapport » :

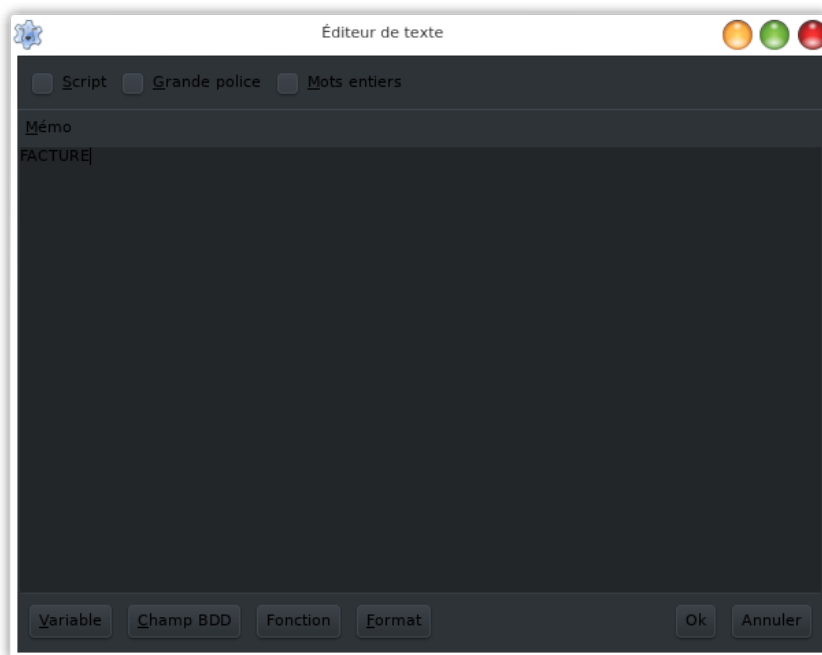




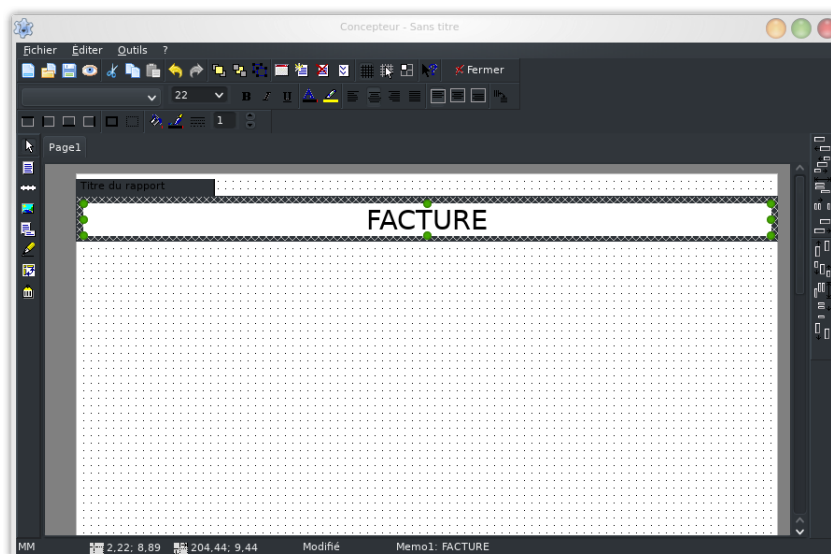
Quand vous avez cliqué sur OK, une (horrible) zone hachurée s'est placée en haut de la page :



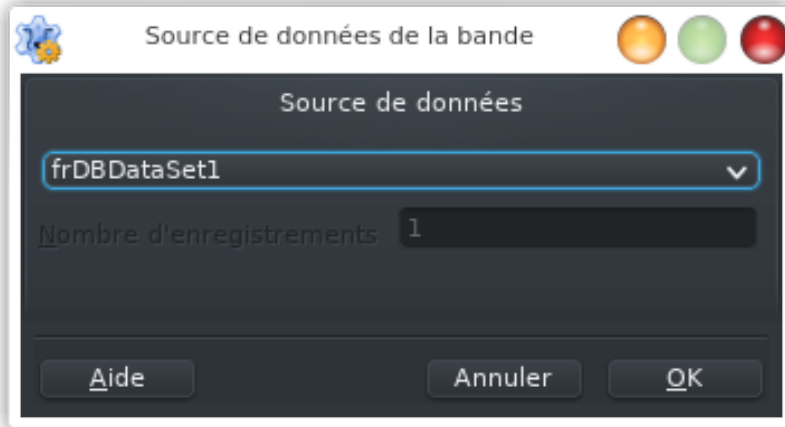
Cliquez ensuite sur la première icône (rectangulaire) de la colonne de gauche, pour placer une zone de texte dans la zone hachurée. Tout de suite, un dialogue vous permet d'écrire du texte :



Inscrivez « FACTURE ». À l'aide des poignées vertes de la zone de texte, centrez le rectangle. Centrez aussi le texte et augmentez la taille de la police, par exemple 22 :



De la même manière, créez une zone hachurée en-dessous de l'en-tête, en choisissant « Données principales » comme type de bande. Un petit dialogue vous demande de sélectionner la source de données : choisissez [frDBDataSet1](#).



À l'aide de la poignée verte inférieure, augmentez la hauteur de la zone hachurée.

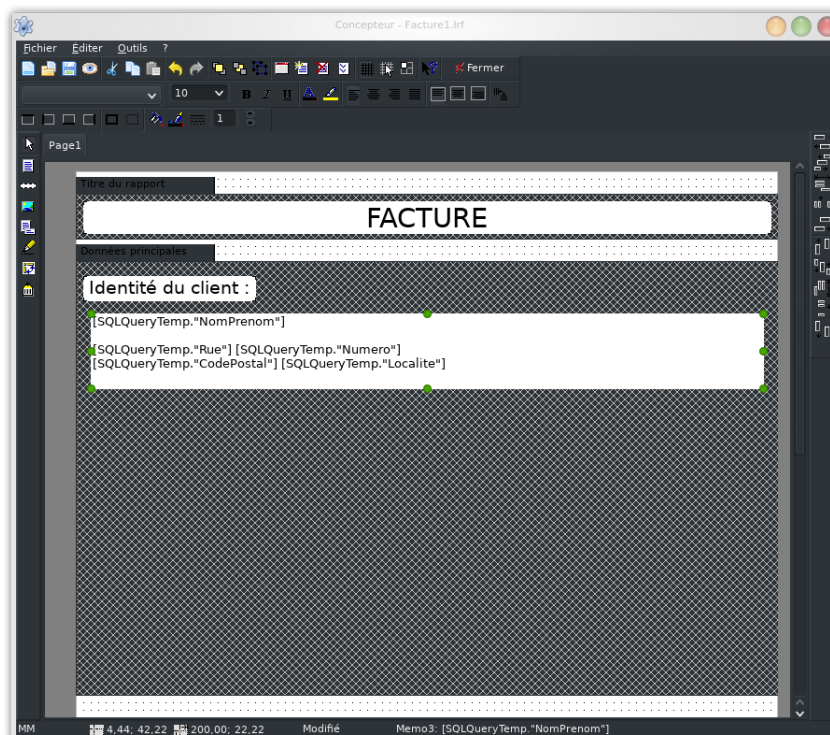
### V-G-3-a - Données statiques

Ajoutez une zone de texte et mettez-y comme titre (taille 14) « Identité du client : ». Pour voir le texte, vous devrez peut-être augmenter la taille du rectangle. En dessous, ajoutez encore une zone dans laquelle vous inscrivez :

```
[SQLQueryTemp."NomPrenom"]  
[SQLQueryTemp."Rue"] [SQLQueryTemp."Numero"]  
[SQLQueryTemp."CodePostal"] [SQLQueryTemp."Localite"]
```

La taille du texte peut être fixée à 10 et l'alignement à gauche.

Les champs du SQLQuery sont mis entre crochets ; c'est leur valeur qui sera affichée. Vous allez toute de suite voir que l'on peut mélanger du texte simple avec les champs.



Créez un second titre « Véhicule loué : », puis un cadre sur toute la largeur, contenant :

```
[SQLQueryTemp."Marque"] [SQLQueryTemp."Modele"]  
Prix par jour : [SQLQueryTemp."Prix"] €
```

Puis créez de nouveau un titre « Location : », et encore un autre cadre sur toute la largeur :

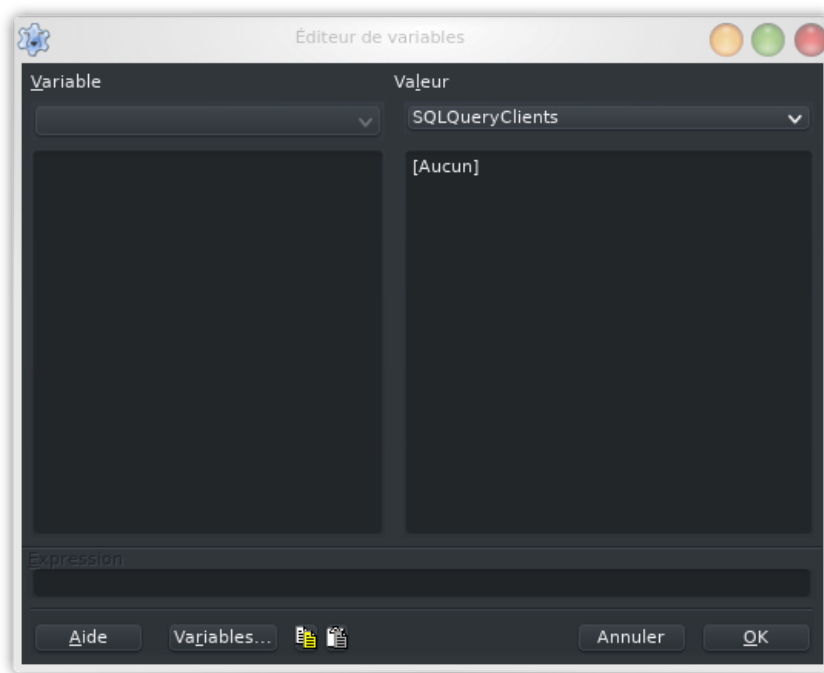
```
Date de début : [SQLQueryTemp."DateDebut"]  
Date de fin prévue : [SQLQueryTemp."DateFin"]  
Rentrée du véhicule : [SQLQueryTemp."DateRentree"]  
Assurance complémentaire : [SQLQueryTemp."Assurance"]
```

### V-G-3-b - Données calculées

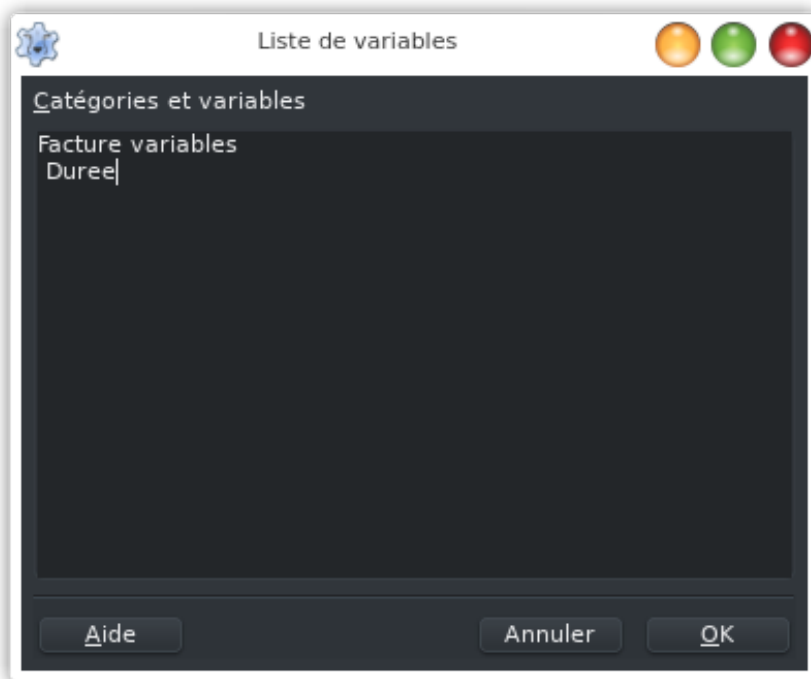
Passons maintenant au plus important pour notre société de location : le pognon ! Ajoutez un quatrième titre « À payer : », puis un rectangle sur toute la largeur.

LazReport possède des fonctions qui permettent de faire des opérations sur des champs de base de données et/ou sur des variables. Malheureusement, il n'existe pas de fonction permettant de calculer le nombre de jours entre les dates de début et de fin de location ; nous allons devoir fournir ce nombre.

Allez dans le menu [Fichier](#) du concepteur et choisissez [Liste des variables](#) :



Cliquez sur le bouton [Variables](#). Dans le dialogue qui suit, inscrivez comme titre « Facture variables » et, à la ligne suivante, **commençant par une espace**, « Duree » :



Cliquez deux fois sur **OK**.

Double-cliquez sur le tout dernier rectangle que vous avez créé sur la largeur de la page, et inscrivez :

```
[Duree] jours * [SQLQueryTemp.Prix] € = [[Duree]*[SQLQueryTemp.Prix]] €
```

C'est à notre application de fournir la valeur de la variable **Duree**, sur demande de LazReport. Fermez le concepteur de rapport et enregistrez la maquette dans le répertoire source de notre application, sous le nom **Facture1.lrf**. Dans l'inspecteur d'objets de l'unité **DataAccess**, sur le composant **frReport1**, cherchez l'événement **OnGetValue**. Cliquez sur les trois points qui l'accompagnent, pour créer une méthode événementielle **frReport1GetValue**.

Voici son contenu :

```
procedure TDataModule1.frReport1GetValue (const ParName : String; var ParValue : Variant);
(* Calcul de la variable Duree sur demande de LazReport *)
begin
    if UpperCase(ParName) = 'DUREE'
    then
        if SQLQueryTemp.FieldByName('DateRentree').AsString <> ''
        then (* Calcul sur la date de rentrée *)
            ParValue := DaysBetween(SQLQueryTemp.FieldByName('DateDebut').AsDateTime,
                                    SQLQueryTemp.FieldByName('DateRentree').AsDateTime) + 1
        else (* Pas de date de rentrée : calcul sur la date de fin prévue *)
            ParValue := DaysBetween(SQLQueryTemp.FieldByName('DateDebut').AsDateTime,
                                    SQLQueryTemp.FieldByName('DateFin').AsDateTime) + 1;
    end;
end;
```

Si la date de rentrée de la voiture est vide, le calcul est fait sur la date de fin de location prévue.

Ce n'est pas tout : si le client a opté pour l'assurance complémentaire, la somme de 10 € est ajoutée au total. Cette donnée étant un booléen, elle est stockée comme une valeur entière 0 ou 1 dans la base de données. C'est parfait pour nous : la somme à éventuellement ajouter au total sera de 10 multiplié par 0 ou 1. Nous complétons ainsi le contenu de notre rectangle :

```
Base : [Duree] jours * [SQLQueryTemp.Prix] € = [[Duree]*[SQLQueryTemp.Prix]] €
Assurance : [10*[SQLQueryTemp."Assurance"]] €
```

Total :  $[[Duree]] * [SQLQueryTemp.Prix] + 10 * [SQLQueryTemp."Assurance"]$  €

## V-G-3-c - Données automatiques

Pour terminer, nous allons vite voir comment rajouter la date du jour dans le rapport. Augmentez la hauteur de la zone hachurée de titre (descendez au besoin la zone hachurée en dessous) et ajoutez un rectangle de texte centré en dessous du titre « FACTURE ». Inscrivez dans ce rectangle :

Date : [DATE]

Facile, non ?

Voici donc la maquette de facture complète :

La maquette de la facture est présentée dans une interface de conception. Elle est structurée comme suit :

- Titre du rapport :** Un rectangle hachuré contenant le mot **FACTURE**.
- Date :** Un rectangle blanc contenant le texte `Date : [DATE]`.
- Données principales :** Une section regroupant plusieurs champs :
  - Identité du client :** Un rectangle blanc contenant les champs `[SQLQueryTemp."NomPrenom"]`, `[SQLQueryTemp."Rue"] [SQLQueryTemp."Numero"]` et `[SQLQueryTemp."CodePostal"] [SQLQueryTemp."Localite"]`.
  - Véhicule loué :** Un rectangle blanc contenant les champs `[SQLQueryTemp."Marque"] [SQLQueryTemp."Modele"]` et `Prix par jour : [SQLQueryTemp."Prix"] €`.
  - Location :** Un rectangle blanc contenant les champs `Date de début : [SQLQueryTemp."DateDebut"]`, `Date de fin prévue : [SQLQueryTemp."DateFin"]`, `Rentrée du véhicule : [SQLQueryTemp."DateRentree"]` et `Assurance complémentaire : [SQLQueryTemp."Assurance"]`.
  - A payer :** Un rectangle blanc contenant les calculs :
    - `Base : [Duree] jours * [SQLQueryTemp.Prix] € = [[Duree]]*[SQLQueryTemp.Prix]] €`
    - `Assurance : [10]*[SQLQueryTemp."Assurance"] €`
    - `Total : [[Duree]]*[SQLQueryTemp.Prix]+10*[SQLQueryTemp."Assurance"] €`

## V-G-4 - Code de création de la facture

Nous allons voir si nous avons bien travaillé. Sur la fiche principale, en dessous du bouton « rentrée de la voiture », ajoutez un dernier bouton « &Facture », nommé `btnFacture`. Le code de la méthode événementielle qui correspond à `OnClick` est le suivant :

```
procedure TMainForm.btnFactureClick (Sender : TObject);
(* Création d'une facture *)
begin
    DataModule1.CreerFacture;
end;
```

C'est en effet logiquement dans le *datamodule* que va figurer le code de création de la facture. Rendez-vous dans l'unité [DataAccess](#).

Dans la section **public** de la déclaration du type **TDataModule1**, ajoutez cette méthode :

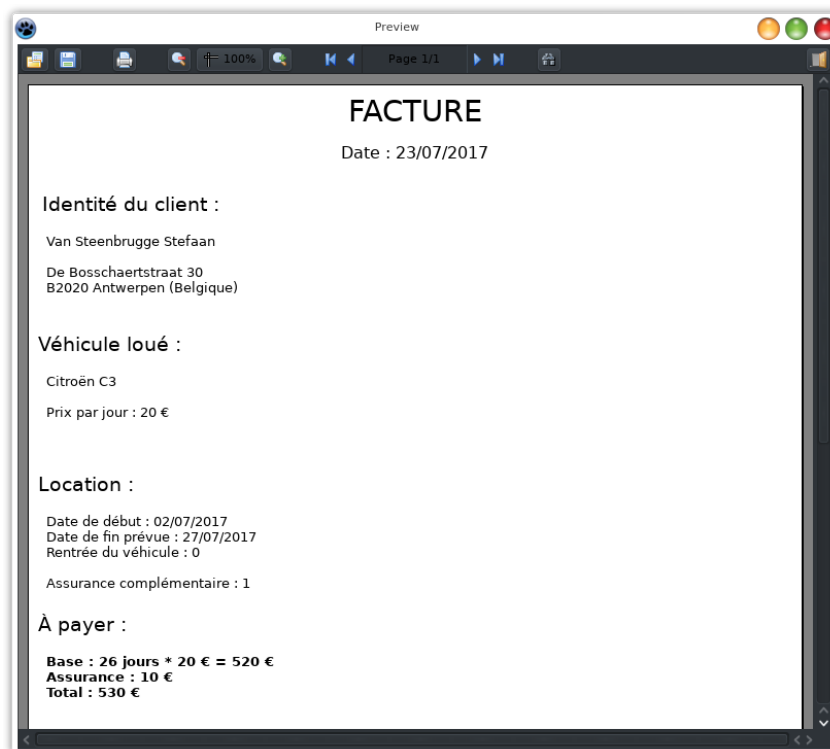
```
procedure CreerFacture;
```

La combinaison de touches **Shift-Ctrl-C**, comme d'habitude, crée son implémentation :

```
procedure TDataModule1.CreerFacture;
(* Remplit l'état Facture1 avec les champs de la location sélectionnée *)
begin
(* Sélection de la location à facturer dans le SQLQueryTemp *)
SQLQueryTemp.Close;
SQLQueryTemp.SQL.Text :=
SQLSyntax.SelectionLocationsFiltre(SQLQueryMain.FieldByName('IdLocation').AsInteger);
SQLQueryTemp.Open;
(* Création du rapport *)
frReport1.LoadFromFile('Facture1.lrf');
frReport1.PrepareReport;
frReport1.ShowReport;
end;
```


La requête de sélection initialise le **SQLQueryTemp** avec les données de la location à facturer, et ce **SQLQuery** sert de source de données au rapport.

Et voici enfin une facture générée par notre application :



Notre application est terminée ! C'est le moment de verser une petite larme, car nous avons bien travaillé. Vous pouvez être fier(e) de vous !

## V-H - Code complet de l'exemple 3

Téléchargez le code source complet de l'application  [ici](#).

## VI - Conclusion

Cet article n'a fait qu'effleurer la création avec Lazarus d'applications utilisant une base de données MySQL. L'exemple de l'application de gestion de société de location de voitures peut vous inspirer dans votre réflexion sur la structuration de vos unités, en séparant clairement l'accès aux données du reste de l'application, en permettant une évolution future vers un autre système de gestion de bases de données, grâce à l'utilisation d'une interface comme cadre pour différentes syntaxes SQL, etc.

Cet exemple d'application souffre de certains défauts, et devrait être amélioré. Vous en avez peut-être repéré certains ; voici les plus graves :

- si vous avez été attentif(ve), dans les dialogues de création et de modification d'une location (classes [TNewLeasingForm](#) et [TUpdateLeasingForm](#)), le rechargement de la *combobox* des voitures, à chaque modification des filtres, devrait être précédé de la désallocation de tous les éléments présents dans la liste, sous peine de petites fuites de mémoire (les éléments qui ne sont pas rechargés dans la liste ne seront pas désalloués à la fermeture du dialogue) ;
- dans le dialogue de modification d'une location ([TUpdateLeasingForm](#)), il faudrait éventuellement adapter la borne inférieure ou supérieure du filtre sur la cylindrée, en fonction de la cylindrée de la voiture louée (sinon cette voiture pourrait être absente de la *combobox*, ce qui déclencherait un plantage).

Ces défauts sont restés tels quels pour ne pas alourdir le code et nécessiter trop d'explications.

## VI-A - Remerciements

Je remercie **Roland Chastain**, **Gilles Vasseur** et **thewolf** pour leur relecture technique, ainsi que **jacques\_jean** pour sa relecture orthographique.