L'accès aux données avec Qt

par Alain Defrance

Date de publication : 27 Août 2008

Dernière mise à jour : 21 Mars 2009

Dans cet article nous verrons comment utiliser la couche d'accès aux données de Qt, comment s'en servir de source de données pour des élements graphiques, et aller plus loin en s'en servant dans la conception d'une application. Cet article s'appuie sur le livre **Qt4 et C++, programmation d'interfaces GUI**

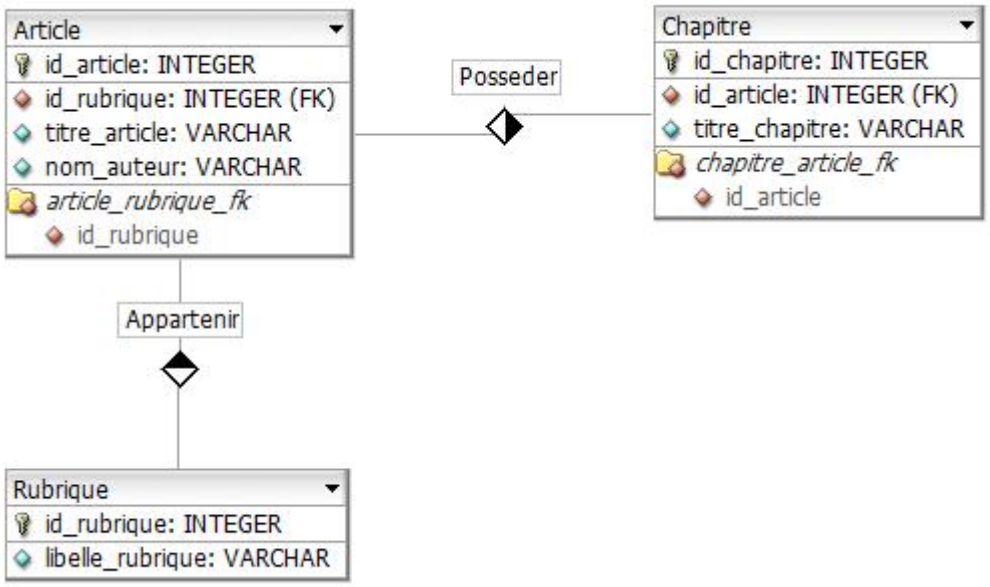
I - Introduction

Beaucoup de frameworks proposent une couche d'accès aux données, c'est-à-dire un système apportant bien souvent une certaine transparence vis-à-vis du **SGBD**.
Nous n'avons plus à nous préoccuper du driver au niveau du code, puisque ce sera le rôle du framework.
Qt en fait partie et nous allons voir quelques-unes des très nombreuses possibilités qu'il offre. Bien évidement, il existe d'autres frameworks, comme **.Net** par exemple, qui propose son célèbre ADO, mais nous utiliserons Qt pour sa facilité d'utilisation et sa portabilité.
Le langage utilisé sera **C++** et nous choisirons comme SGBD SQL-Server.
Afin de pouvoir suivre ce tutoriel il sera nécessaire de savoir utiliser Qt (notamment le designer et connaître le rôle du *.pro*). Vous trouverez toutes les informations utiles dans **la rubrique Qt**.
Les sources finales sont accessibles **ici**.

II - La base de données

II-A - Le MCD (Modèle conceptuel de données)

Afin d'y voir clair nous nous servirons un MCD, voici donc l'organisation des données:



Modèle conceptuel de données

Selon les règles de l'art, identifier l'entité Chapitre relativement à l'entité Article est sémantiquement plus pertinent, mais la conception des bases de données n'est pas le sujet de l'article, et il est tout à fait envisageable d'utiliser ce modèle.

II-B - Le script SQL de création

Le script SQL de création avec SQL-Server

```
CREATE TABLE Rubrique
(
 id_rubrique INT NOT NULL,
 libelle_rubrique VARCHAR(50)
);
```

Le script SQL de création avec SQL-Server

```
CREATE TABLE Article
(
 id_article INT NOT NULL,
 titre_article VARCHAR(50) NOT NULL,
 nom_auteur VARCHAR(30) NOT NULL,
 id_rubrique INT NOT NULL
);

CREATE TABLE Chapitre
(
 id_chapitre INT NOT NULL,
 titre_chapitre VARCHAR(50) NOT NULL,
 id_article INT NOT NULL
);

ALTER TABLE Rubrique
ADD
 CONSTRAINT PK_Rubrique PRIMARY KEY (id_rubrique);

ALTER TABLE Article
ADD
 CONSTRAINT PK_Article PRIMARY KEY (id_article),
 CONSTRAINT FK_Article_Rubrique FOREIGN KEY (id_rubrique) REFERENCES Rubrique (id_rubrique) ON DELETE
 CASCADE;

ALTER TABLE Chapitre
ADD
 CONSTRAINT PK_Chapitre PRIMARY KEY (id_chapitre),
 CONSTRAINT FK_Chapitre_Article FOREIGN KEY (id_article) REFERENCES Article (id_article) ON DELETE
 CASCADE;
```

III - L'accès aux données

Si au niveau de l'utilisation le choix du SGBD est transparent, il n'en est pas de même à bas niveau.
C'est pour cela que différents drivers sont utilisables au niveau de la connexion, mais pour nous, développeurs, nous n'avons pas à nous préoccuper de ce qui se passe à si bas niveau d'abstraction. Il est cependant utile de connaître quels drivers existent, et comment les utiliser.
Certains sont disponibles dans la version open source, d'autres nécessitent une recompilation de Qt.

III-A - Les drivers disponibles

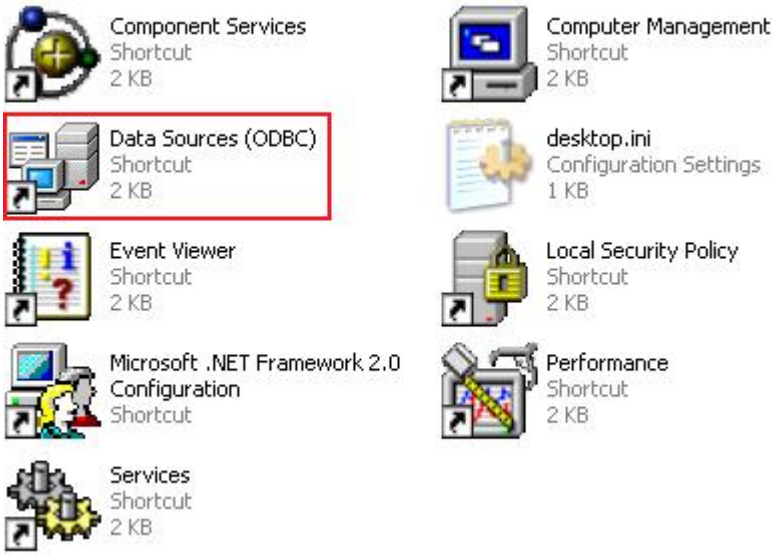
Pilote	Base de données	Disponible en open source
QDB2	IBM DB2 version 7.1 et ultérieure	Non
QIBASE	Borland InterBase	Non
QMYSQL	MySQL	Non
QOCI	Oracle (Oracle Call Interface)	Oui
QODBC	ODBC (inclut Microsoft SQL Server)	Oui
QPSQL	PostgreSQL versions 6.x et 7.x	Non
QSQLITE	SQLite version 3 et ultérieure	Oui
QSQLITE2	SQLite version 2	Non
QTDS	Sybase Adaptive Server	Non

*On remarque qu'il y a peu de drivers disponibles en version open source, **ODBC** en fait partie.*

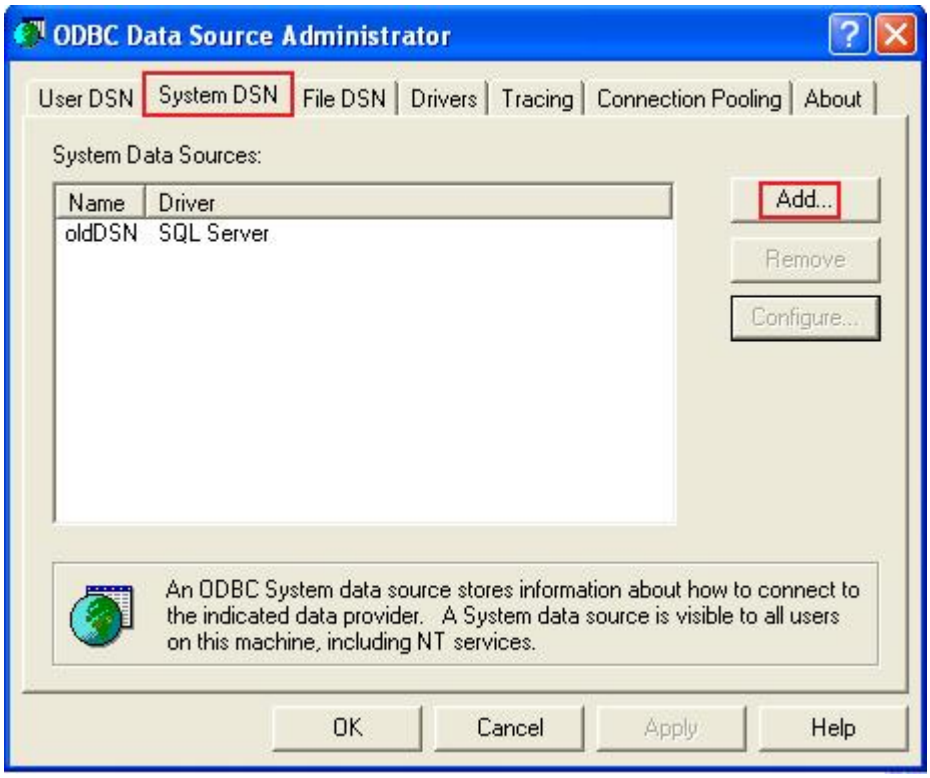
*Au niveau des connexions à Oracle, le driver **QOpenOCCI** basé sur la librairie open source **OCILIB** est réputé comme bien plus performant, son utilisation sera donc à privilégier en production.*
L'utilisation d'ODBC en production est suicidaire en terme de performance, mais ici dans le cadre d'un apprentissage, il est tout à fait possible d'utiliser un SGBD autre que SQL Server avec ODBC comme middleware.

III-B - La création d'un DSN (uniquement pour ODBC)

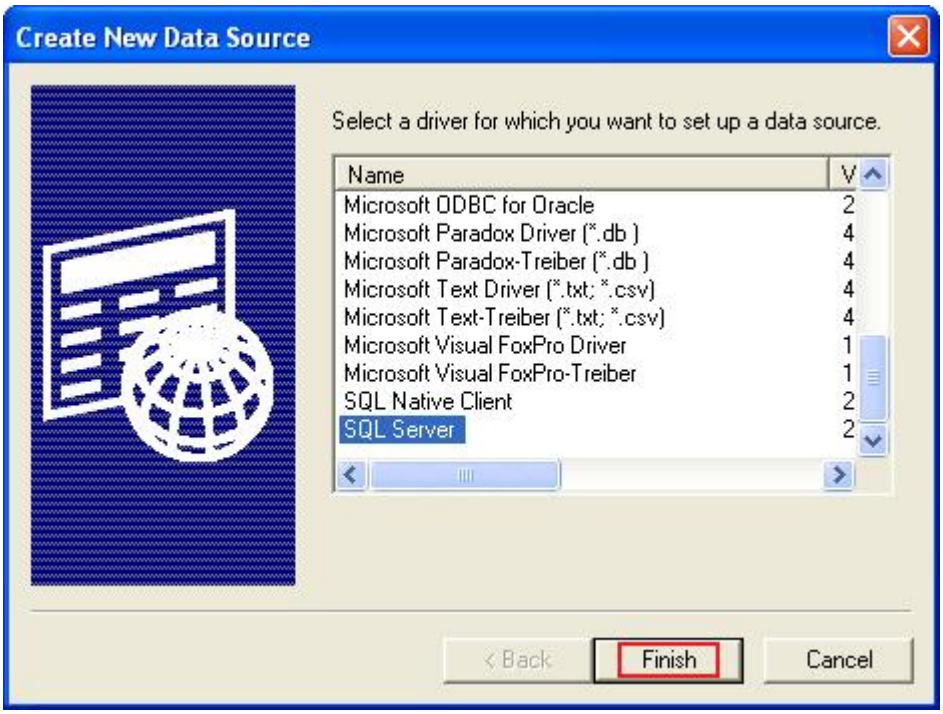
Un DSN est une source de données ODBC, il permet principalement d'identifier une connexion et de lui associer un certain SGBD.
Sous windows XP, la configuration des sources de données ODBC est accessible par :
Avec l'affichage par catégories :
Démarrer/Panneau de configuration/Performances et maintenance/Outils d'administration/
Avec l'affichage classique :
Démarrer/Panneau de configuration/Outils d'administration/



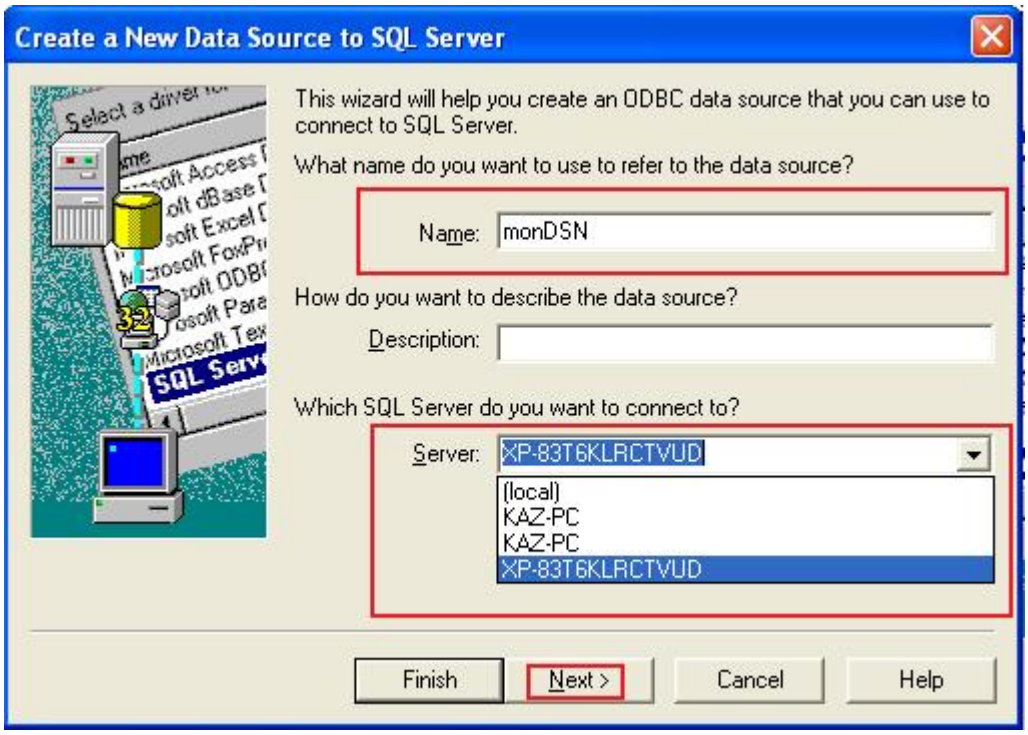
Accès aux sources de données



Dans l'onglet 'Source de données système', cliquez sur 'Ajouter'

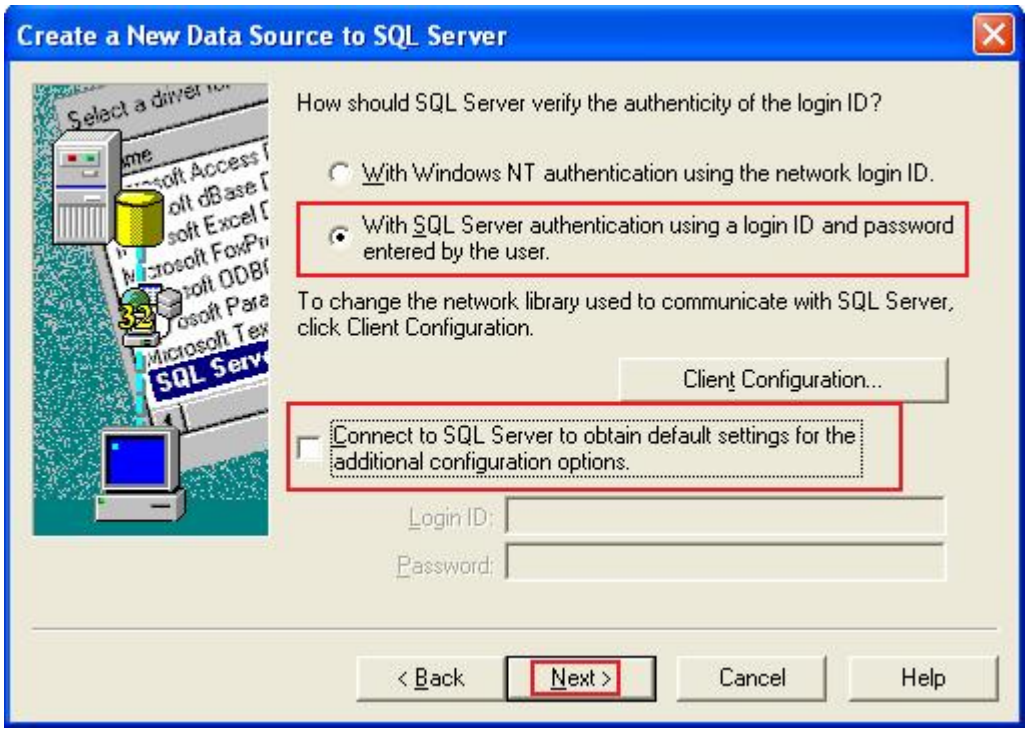


Après avoir choisi le SGBD (dans notre cas SQL Server), cliquez sur 'Terminer'

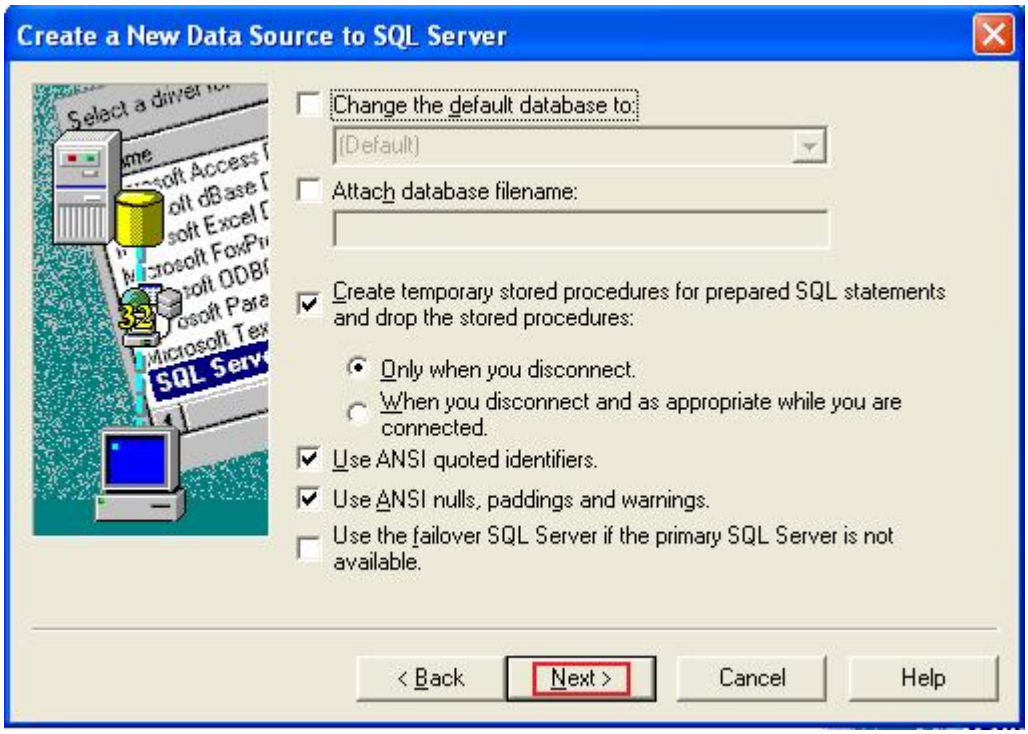


Saisir le nom (qui servira d'identifiant de la source de données), choisir le serveur (présent dans le menu déroulant), puis cliquez sur 'Suivant >'

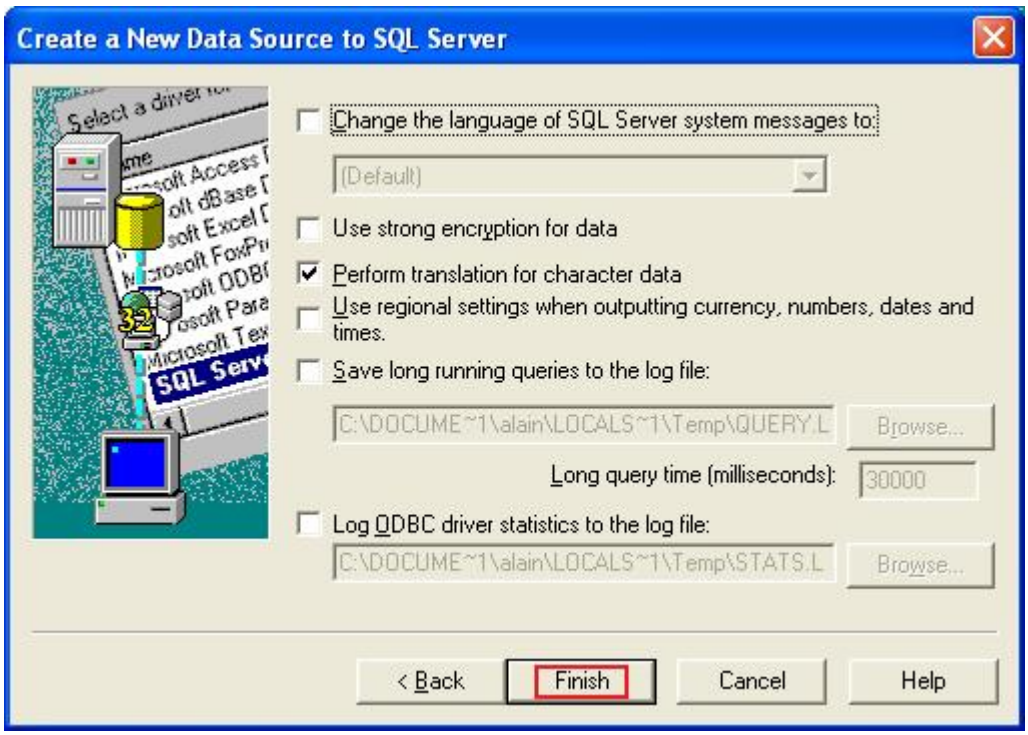
*Il est possible de spécifier directement une **adresse IP**, ou une adresse **FQDN** comme nom de serveur.*



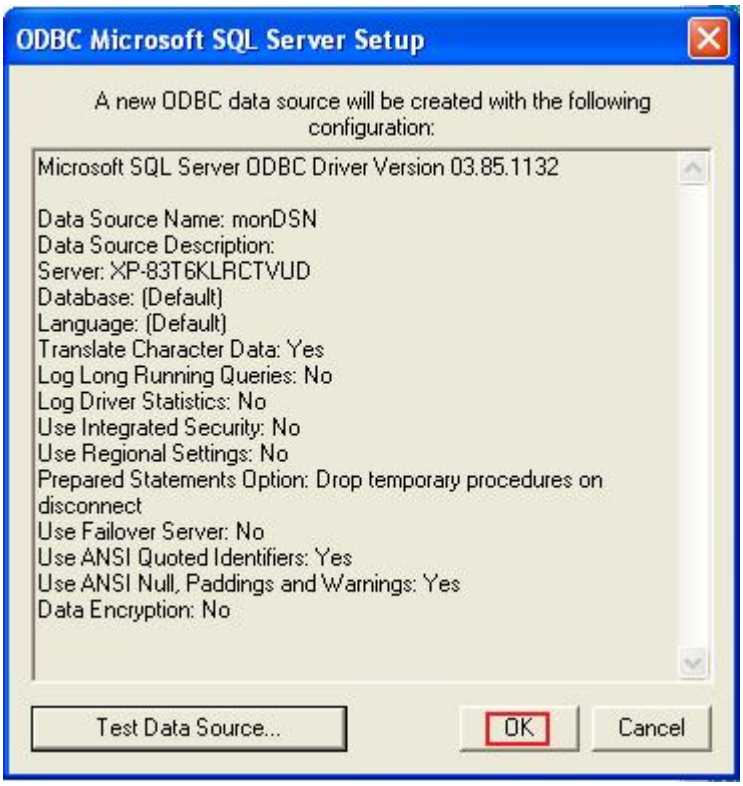
Activer l'authentification SQL Server, et décocher la récupération de la configuration par défaut, puis cliquer sur 'Suivant >'



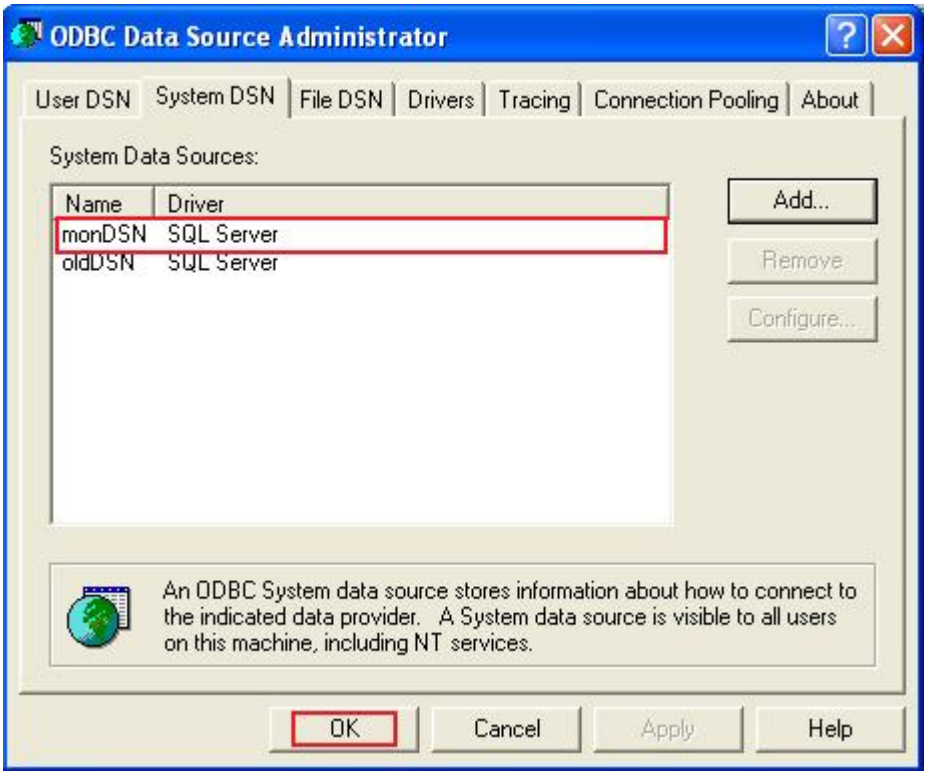
Ne rien toucher et cliquer sur 'Suivant >'



Ne rien toucher et cliquer sur 'Terminer'



Inutile de tester la source de données cela ne marchera pas car nous n'avons pas précisé d'identifiant et de mot de passe, cliquez sur 'OK'



Notre nouvelle source de données est bien présente, cliquez sur 'OK', notre système est maintenant prêt à faire le lien entre QODBC et SQL Server

III-C - Connexion à la base de données

Avant toute chose nous devons modifier le *.pro* pour que les classes d'accès aux données soient accessibles. Pour ce faire il suffit d'ajouter ceci au *.pro* :

application.pro

```
QT += sql
```

Nous allons maintenant utiliser notre lien ODBC fraîchement créé.
Les connexions s'utilisent au travers de la classe **QSqlDatabase**.
Cette classe possède une méthode statique **QSqlSatabase::addDatabase(const QString)** renvoyant une instance de **QSqlDatabase** et reçoit en paramètre une chaîne de caractères correspondant au driver utilisé.

Notre première connexion

```
#include <QApplication>
#include <QSqlDatabase>
#include <QSqlError>
#include <QMessageBox>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    win_Form *window = new win_Form();

    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
    db.setDatabaseName("monDSN"); // DSN que nous venons de créer.
    db.setUserName("alain.defrance");
    db.setPassword("plop");

    if(!db.open())
    {
        QMessageBox::critical(0, QObject::tr("Database Error"), db.lastError().text());
    }

 window->show();
    return app.exec();
}
```

Si aucun message d'erreur n'apparaît, alors la connexion s'est bien déroulée.
Notez que nous avons utilisé une instance de type win_Form qui est un type dérivé d'une fenêtre générée avec le designer (nous suivons ainsi les recommandations de Trolltech). L'interface graphique nous permet de voir que l'application s'est bien exécutée et d'apporter une touche visuel. Nous verrons par la suite comment lier les données directement à l'interface graphique.

III-D - Requête sans retour de données

Il y a deux différences fondamentales dans les requêtes **SQL**, celles qui renvoient des données, et celles qui n'en revoient pas.
La différence au niveau applicatif est qu'il faudra (ou pas) gérer un retour de données.
Dans certains frameworks (.Net par exemple), il existe plusieurs fonctions permettant d'exécuter des requêtes, certaines pour gérer des retours, d'autres non ... Cette utilisation peut paraître peu pratique aux yeux de certains. En effet, le programmeur peut se tromper de fonction et provoquer une erreur, alors que la requête est correcte.
Qt procède différement dans le sens où il n'existe qu'un seul moyen d'envoyer la requête. Libre à nous de parcourir le recordset si on souhaite récupérer les valeurs de retour. On utilise pour cela une instance de la classe **QSqlQuery** en appelant sa méthode **QSqlQuery::exec(const QString)**.

Simple insertion de données

```
#include <QApplication>
#include <QSqlDatabase>
#include <QMessageBox>
#include <QSqlError>
#include <QSqlQuery>
#include "win_main.hpp"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    win_Form *window = new win_Form();

    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
    db.setDatabaseName("monDSN");
    db.setPassword("plop");
    db.setUserName("alain.defrance");

    if(!db.open())
    {
        QMessageBox::critical(0, QObject::tr("Database Error"), db.lastError().text());
    }

    QSqlQuery requeteur;

 requeteur.exec("INSERT INTO Rubrique(id_rubrique, libelle_rubrique) VALUES (1, 'nouvelle rubrique')");

    window->show();
    return app.exec();
}
```

III-E - Requête avec retour de données.

Comme expliqué dans la partie précédente, la manière de requêter sera la même.
Nous avons seulement à parcourir le recordset afin de récupérer et traiter un à un les tuples retournés par la requête.

Simple insertion de données

```
#include <QApplication>
#include <QSqlDatabase>
#include <QMessageBox>
#include <QSqlError>
#include <QSqlQuery>
#include "win_main.hpp"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    win_Form *window = new win_Form();

    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
    db.setDatabaseName("monDSN");
    db.setPassword("plop");
    db.setUserName("alain.defrance");

    if(!db.open())
    {
        QMessageBox::critical(0, QObject::tr("Database Error"), db.lastError().text());
    }

    QSqlQuery requeteur;
    requeteur.exec("DELETE FROM Rubrique");

 requeteur.exec("INSERT INTO Rubrique(id_rubrique, libelle_rubrique) VALUES (1, 'nouvelle rubrique')");
    requeteur.exec("SELECT * FROM Rubrique");

    while(requeteur.next())
```

Simple insertion de données

```
    {
        int id_rubrique = requeteur.value(0).toInt();
        QString libelle_rubrique = requeteur.value(1).toString();
        QMessageBox::information(
            0,
            QObject::tr("Information récupérée"),
            "Id : " + QString::number(id_rubrique) + "\nLibellé : " + libelle_rubrique
            );
    }

    window->show();
    return app.exec();
}
```

Ici nous effectuons trois requêtes :

- la première supprime tous les tuples présents dans la table *rubrique*
- la seconde ajoute une nouvelle rubrique
- la troisième récupère tous les tuples présents dans la table *rubrique*

Bien évidement dans notre cas il y aura toujours un seul tuple à retourner à chaque exécution, puisque nous vidons la base de données à chaque lancement du programme.
Si c'est son premier appel la méthode **QSqlQuery::next()** place la lecture au premier enregistrement retourné, sinon elle positionnera la lecture à l'enregistrement suivant.

IV - Les variables bind

IV-A - Pourquoi des variables bind

IV-A-1 - Raisons de sécurité

Si vous êtes habitués au développement avec SGBD, vous êtes probablement sensibles aux risques **d'injections SQL**. Les variables bind ont pour particularité de traiter ces attaques afin d'éviter au programmeur de se préoccuper de cela. Nous évitons ainsi une surcharge de travail, et gagnions en fiabilité.

IV-A-2 - Raisons de lisibilité

Lorsque nous développons une application et que nous manipulons des données, nous le faisons au travers de variables. Afin de les réutiliser pour construire une requête SQL, la manière la plus instinctive est de la construire par concaténation.
Nous obtenons ainsi un code similaire à celui-ci :

Construction d'un requête SQL

```
QString nomRubrique;
QString maRequete;


// Ici on valorise la variable nomRubrique (avec l'interface homme-machine par exemple) //


// Puis nous construisons la requête
maRequete = "INSERT INTO Rubrique(id_rubrique, libelle_rubrique) VALUES (1, '" + nomRubrique + "')";
```

Notez que nous commençons déjà à perdre en visibilité avec une seule variable, alors avec une dizaine de valeurs cela deviendra complètement illisible.

IV-A-3 - Raisons d'optimisation

Pour comprendre le véritable interêt des variables bind, il est indispensable de comprendre comment un SGBD gère l'optimisation des requêtes.
Lorsqu'un SGBD reçoit une requête à exécuter, il doit tout d'abord l'évaluer, c'est-à-dire la découper, l'analyser, pour ensuite l'exécuter. Cette évaluation peut être simple comme très complexe. Il convient donc de stocker le résultat de cette analyse afin d'éviter d'évaluer deux fois la même requête.
On considère que deux requêtes sont identiques si tout simplement elles sont strictement égales, **au caractère près**. Cela implique que lorsque deux requêtes ont le même but (un certain type d'insertion, récupérer les articles d'une certaine rubrique, etc ...), elles sont toutes évaluées à leur tours.
Prenons l'exemple de ces deux insertions :

Deux requêtes d'insertion

```
INSERT INTO Rubrique(id_rubrique, libelle_rubrique) VALUES (1, 'rubrique 1');
INSERT INTO Rubrique(id_rubrique, libelle_rubrique) VALUES (2, 'rubrique 2');
```

Aussi surprenant que cela puisse paraître, le SGBD analysera chaque requête comme si elle n'avait rien à voir car le caractère '1' est différent du caractère '2'.

IV-B - Qu'est-ce qu'une variable bind, comment les utilise-t-on ?

Tout d'abord une variable bind est une variable comme une autre mis à part qu'elle est présente directement dans le SQL et que nous allons lui associer une variable applicative.
Les différents SGBD utilisent la syntaxe de leur choix. Quant à Qt, il supporte la syntaxe d'Oracle et SQL-Server.
Dans ce tutoriel, nous utiliserons la syntaxe d'Oracle, qui est plus lisible car les variables y sont nommées.
Sans plus attendre, voici une requête utilisant des variables bind.

Utilisation simple d'une variable bind

```
INSERT INTO Rubrique(id_rubrique, libelle_rubrique) VALUES (:id, :rubrique);
```

*Notez que la seule différence entre un nom de champ et une variable bind, est qu'une variable bind est précédée du caractère **:***

Il nous faut par la suite associer à cette variable bind une variable applicative. Cela se fait grâce à la méthode **QSqlQuery::bindValue(const QString, const QVariant)**.
La requête n'est alors plus passée à **QSqlQuery::exec(const QString)**, mais à la méthode **QSqlQuery::prepare(const QString)**.
Pour envoyer la requête il suffit d'appeler la méthode **QSqlQuery::exec()**.
Afin de clarifier cette nouvelle possibilité, nous allons reprendre notre premier exemple, mais cette fois avec des variables blind.

Simple insertion de données

```
#include <QApplication>
#include <QSqlDatabase>
#include <QMessageBox>
#include <QSqlError>
#include <QSqlQuery>
#include "win_main.hpp"

int main(int argc, char *argv[])
```

Simple insertion de données

```
{
    QApplication app(argc, argv);
    win_Form *window = new win_Form();

    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
    QString nouvelleRubrique = "nouvelle rubrique"; // Variable applicative
    QString nouvelleRubrique2 = "nouvelle rubrique 2"; // Variable applicative

    db.setDatabaseName("monDSN");
    db.setPassword("plop");
    db.setUserName("alain.defrance");

    if(!db.open())
    {
        QMessageBox::critical(0, QObject::tr("Database Error"), db.lastError().text());
    }

    QSqlQuery requeteur;
    requeteur.exec("DELETE FROM Rubrique");



 requeteur.prepare("INSERT INTO Rubrique(id_rubrique, libelle_rubrique) VALUES (:newId, :newRubrique)");
     // Utilisation de variable bind

 requeteur.bindValue(":newId", 1); // Liaison entre la variable bind et la variable applicative pour l'identifian
    requeteur.bindValue(":newRubrique",
 nouvelleRubrique); // Liaison entre la variable bind et la variable applicative pour le titre de la rubrique.
    requeteur.exec();
    requeteur.bindValue(":newId", 2);
    requeteur.bindValue(":newRubrique", nouvelleRubrique2);
    requeteur.exec();


    requeteur.exec("SELECT * FROM Rubrique");

    while(requeteur.next())
    {
        int id_rubrique = requeteur.value(0).toInt();
        QString libelle_rubrique = requeteur.value(1).toString();
        QMessageBox::information(
            0,
            QObject::tr("Information récupérée"),
            "Id : " + QString::number(id_rubrique) +
            "\nLibellé : " + libelle_rubrique
            );
    }

    window->show();
    return app.exec();
}
```

Notez que lorsque l'on exécute deux fois une même requête il est inutile d'appeler une seconde fois le prepare, il suffit juste de lier la variable bind avec une nouvelle valeur, puis appeler la méthode ***QSqlQuery::exec()****.*

V - Les modèles

Nous avons vu comment interroger et alimenter la base de données, mais si nous voulons par exemple lier ces données à une interface graphique, il n'est pas possible d'avoir un réel lien entre nos données et notre composant graphique.
Cependant il existe la classe **QSqlTableModel** qui permet d'interroger, alimenter, et lier les données (à un QTableView) par exemple, chose que nous verrons un peu plus tard. Pour le moment nous allons voir comment interroger et alimenter la base de données au travers de cette classe.

V-A - Alimenter la base de données

Avant toute chose il faut appeler la méthode **QSqlTableModel::setTable(const QString)** pour définir la table d'insertion.
La méthode **QSqlTableModel::insertRows(int, int)** permet d'ajouter un nouvel enregistrement, il suffit par la suite de saisir les données avec la méthode **QSqlTableModel::setData(const QModelIndex, const QVariant)**

Insertion avec un modèle

```
#include <QApplication>
#include <QSqlDatabase>
#include <QSqlError>
#include <QSqlRecord>
#include <QSqlQuery>

#include "win_form.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    win_Form *window = new win_Form();

    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");

    db.setDatabaseName("monDSN");
    db.setPassword("plop");
    db.setUserName("alain.defrance");


    QSqlTableModel model;
    model.setTable("Rubrique");
    for(int i = 0; i < 3; ++row)
    {
        model.insertRows(i, 1);
        model.setData(model.index(i, 0), i);
        model.setData(model.index(i, 1), "rubrique avec model " + QString::number(i));
        model.submitAll();
    }

    window->show();
    return app.exec();
}
```

V-B - Interroger la base de données

L'interrogation de la base de données se fait au travers d'un modèle. Nous pouvons appliquer un filtre qui n'est autre qu'une sélection (utilisation de la clause WHERE).
Nous n'avons donc plus à écrire nos requêtes SQL, le modèle le fera pour nous. Pour appliquer un filtre nous utiliserons la méthode **QSqlTableModel::setFilter(const QString)**, nous exécuterons la requête au travers de la méthode **QSqlTableModel::select()**.
Une fois la requête effectuée, le modèle est porteur de données que nous devons extraire au travers de la méthode **QSqlQueryModel::record(int)** revoyant un **QSqlRecord**, directement accessible depuis notre modèle puisque **QSqlTableModel** hérite de **QSqlQueryModel**. A partir de notre QSqlRecord nous disposons de deux nouvelles méthodes renvoyant un **QVariable** : **QSqlRecord::value(int)** et **QSqlRecord(const QString)**.
Afin de mieux comprendre je vous propose de regarder cet exemple :

Iterrogation avec un modèle

```
#include <QApplication>
#include <QSqlDatabase>
#include <QSqlError>
#include <QSqlRecord>
#include <QSqlQuery>
```

Interrogation avec un modèle

```
#include "win_form.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    win_Form *window = new win_Form();

    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");

    db.setDatabaseName("monDSN");
    db.setPassword("plop");
    db.setUserName("alain.defrance");


    QSqlTableModel model;
    model.setTable("Rubrique");

 model.setFilter("libelle_rubrique LIKE '% 2'"); // équivalent à SELECT * FROM Rubrique WHERE libelle_rubrique LI
    model.select();
    for(int i = 0; i < model.rowCount(); ++i)
    {
        QSqlRecord record = model.record(i);
        QMessageBox::information(
            0,
            QObject::tr("Information récupérée"),
            "Id : " + QString::number(record.value(0).toInt()) +
            "\nLibellé : " + record.value(1).toString()
            );
    }

    window->show();
    return app.exec();
}
```

Pour des questions d'optimisation il est préférable de récupérer les valeurs des champs par indice, comme utilisé dans l'exemple ci-dessus, et non pas par nom. En effet une recherche de l'indice sera faite en interne à chaque récupération par nom.
Pour régler ce problème tout en conservant une clareté dans le code, nous pouvons utiliser la méthode **QSqlRecord::indexOf(const QString)**

Interrogation avec un modèle

```
QSqlTableModel model;
model.setTable("Rubrique");
model.setFilter("libelle_rubrique LIKE '% 2'"); // équivalent à SELECT * FROM Rubrique WHERE libelle_rubrique LIK
model.select();
int idIndex = model.record().indexOf("id_rubrique");
int libIndex = model.record().indexOf("libelle_rubrique");
for(int i = 0; i < model.rowCount(); ++i)
{
    QSqlRecord record = model.record(i);
    QMessageBox::information(
        0,
        QObject::tr("Information récupérée"),
        "Id : " + QString::number(record.value(idIndex).toInt()) +
        "\nLibellé : " + record.value(libIndex).toString()
        );
}
```

V-C - Modifier les données

La modification est une combinaison d'une interrogation et d'une alimentation. Pour ce faire nous appelons la méthode **QSqlRecord::setValue(const QString, const QVariant)**, le premier paramètre est le nom du champ à modifier, et

le second est la nouvelle valeur. Comme pour la lecture des données, il existe **QSqlRecord::setValue(int, const QVariant)**. Cette dernière est, comme pour l'accès aux données, plus rapide que son homologue.
Après avoir modifié le record, il suffit d'assigner le nouvel enregistrement grâce à la méthode **QSqlTableModel::setRecord(int, QSqlRecord)** puis effectuer l'enregistrement.

Modification avec un modèle

```
model.setFilter("libelle_rubrique LIKE '% 2'"); // équivalent à SELECT * FROM Rubrique WHERE libelle_rubrique LIK
model.select();
int idIndex = model.record().indexOf("id_rubrique");
int libIndex = model.record().indexOf("libelle_rubrique");
for(int i = 0; i < model.rowCount(); ++i)
{
    QSqlRecord record = model.record(i);
    record.setValue(libIndex, record.value(libIndex).toString() + " modifié");
    model.setRecord(i, record);
    QMessageBox::information(
        0,
        QObject::tr("Information récupérée"),
        "Id : " + QString::number(record.value(idIndex).toInt()) +
        "\nLibellé : " + record.value(libIndex).toString()
        );
}
```

V-D - Supprimer les données

La suppression des données est plutôt simple puisqu'elle repose sur l'appel de la méthode **QSqlTableModel::removeRows(int, int)**, le premier paramètre est l'index de l'enregistrement à supprimer et le second le nombre d'enregistrements à supprimer.

Suppression avec un modèle

```
// On insert une rubrique 0, 1 et 2.
QSqlTableModel model;
model.setTable("Rubrique");
for(int row = 0; row < 3; ++row)
{
    model.insertRows(row, 1);
    model.setData(model.index(row, 0), row);
    model.setData(model.index(row, 1), "rubrique avec model " + QString::number(row));
    model.submitAll();
}

// On applique un filtre pour selectionner la rubrique 2 puis on supprime tous les enregistrements correspondants
model.setFilter("libelle_rubrique LIKE '% 2'");
model.select();
if(model.rowCount() > 0)
{
    model.removeRows(0, model.rowCount());
    model.submitAll();
}

// On selectionne les enregistrements présents en base de données afin de vérifier que l'on a bien supprimer la r
QSqlTableModel modelView;
modelView.setTable("Rubrique");
modelView.select();
for(int i = 0; i < modelView.rowCount(); ++i)
{
    QSqlRecord record = modelView.record(i);
    QMessageBox::information(
        0,
        QObject::tr("Information récupérée"),
        "Id : " + QString::number(record.value(0).toInt()) +
        "\nLibellé : " + record.value(1).toString()
        );
}
```

Suppression avec un modèle

Au travers de ces diverses manipulations, il est plus facile de comprendre le rôle des modèles. Ces modèles ne sont en quelque sorte qu'une copie de la base de données. Lors de l'appel de la méthode ***QSqlTableModel::select()*** *une copie des données se fait de la base de données vers le modèle. Il convient alors de travailler directement sur le modèle puis de mettre à jour les nouvelles données vers le SGBD avec la méthode* ***QSqlTableModel::submitAll()****.*

VI - Les modèles avec l'interface homme-machine

Nous avons déjà fait le plus dur en apprenant comment utiliser les données d'un SGBD quelconque. Il ne reste plus qu'à apprendre à réutiliser ces modèles avec l'interface graphique. Bien entendu Trolltech a prévu cela, et notre tâche sera grandement simplifiée. Non seulement il est possible de réutiliser ces modèles, mais il est possible de déléguer à l'interface graphique la gestion de ces derniers. Nous verrons aussi qu'il est possible de mettre en place un lien entre les composants graphiques afin d'avoir une organisation maître/détail représentant des liens relationnels.

VI-A - Le jeu d'essai

Nous allons commencer par mettre un peu d'ordre dans la base de données que nous avons utilisée pour nos tests, puis nous allons la remplir avec un petit jeu d'essai.

Jeu d'essai

```
DELETE FROM Rubrique;
DELETE FROM Article;
DELETE FROM Chapitre;

INSERT INTO Rubrique(id_rubrique, libelle_rubrique) VALUES (1, 'rubrique A');
INSERT INTO Rubrique(id_rubrique, libelle_rubrique) VALUES (2, 'rubrique B');
INSERT INTO Rubrique(id_rubrique, libelle_rubrique) VALUES (3, 'rubrique C');

INSERT INTO Article(id_article, titre_article, nom_auteur, id_rubrique) VALUES
 (1, 'article a', 'auteur i', 1);
INSERT INTO Article(id_article, titre_article, nom_auteur, id_rubrique) VALUES
 (2, 'article b', 'auteur h', 1);
INSERT INTO Article(id_article, titre_article, nom_auteur, id_rubrique) VALUES
 (3, 'article c', 'auteur g', 1);
INSERT INTO Article(id_article, titre_article, nom_auteur, id_rubrique) VALUES
 (4, 'article d', 'auteur f', 2);
INSERT INTO Article(id_article, titre_article, nom_auteur, id_rubrique) VALUES
 (5, 'article e', 'auteur e', 2);
INSERT INTO Article(id_article, titre_article, nom_auteur, id_rubrique) VALUES
 (6, 'article f', 'auteur d', 2);
INSERT INTO Article(id_article, titre_article, nom_auteur, id_rubrique) VALUES
 (7, 'article g', 'auteur c', 3);
INSERT INTO Article(id_article, titre_article, nom_auteur, id_rubrique) VALUES
 (8, 'article h', 'auteur b', 3);
INSERT INTO Article(id_article, titre_article, nom_auteur, id_rubrique) VALUES
 (9, 'article i', 'auteur a', 3);

INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (1, 'chapitre 1 de l''article a', 1);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (2, 'chapitre 2 de l''article a', 1);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (3, 'chapitre 3 de l''article a', 1);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (4, 'chapitre 1 de l''article b', 2);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (5, 'chapitre 2 de l''article b', 2);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (6, 'chapitre 3 de l''article b', 2);
```

Jeu d'essai

```
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (7, 'chapitre 1 de l''article c', 3);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (8, 'chapitre 2 de l''article c', 3);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (9, 'chapitre 3 de l''article c', 3);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (10, 'chapitre 1 de l''article d', 4);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (11, 'chapitre 2 de l''article d', 4);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (12, 'chapitre 3 de l''article d', 4);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (13, 'chapitre 1 de l''article e', 5);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (14, 'chapitre 2 de l''article e', 5);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (15, 'chapitre 3 de l''article e', 5);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (16, 'chapitre 1 de l''article f', 6);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (17, 'chapitre 2 de l''article f', 6);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (18, 'chapitre 3 de l''article f', 6);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (19, 'chapitre 1 de l''article g', 7);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (20, 'chapitre 2 de l''article g', 7);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (21, 'chapitre 3 de l''article g', 7);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (22, 'chapitre 1 de l''article h', 8);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (23, 'chapitre 2 de l''article h', 8);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (24, 'chapitre 3 de l''article h', 8);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (25, 'chapitre 1 de l''article i', 9);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (26, 'chapitre 2 de l''article i', 9);
INSERT INTO Chapitre(id_chapitre, titre_chapitre, id_article) VALUES
 (27, 'chapitre 3 de l''article i', 9);
```

VI-B - Lier les modèles aux composants graphiques

A partir de maintenant nous allons écrire notre code dans le constructeur de notre fenêtre afin d'avoir accès rapidement à nos composants graphiques et de déléguer la responsabilité des données à notre fenêtre.
L'utilisation du modèle ne diffère pas et nous avons seulement à lier la source de données (le modèle), à notre composant. Pour lier à un QTableView il suffit d'appeler la méthode héritée **QAbstractItemView::setModel(QAbstractItemModel *)**. Nous pouvons appeler d'autres méthodes permettant de masquer certaines colonnes, définir un critère de tri, et bien d'autres choses non abordées ici.

Lier le modèle à un QTableView

```
win_Form::win_Form(QWidget *parent)
    : QWidget(parent), ui(new Ui::win_FormClass)
{
 // On défini des constances pour simplifier l'utilisation des indices de colonne.
    enum {
        RUBRIQUE_ID = 0,
        RUBRIQUE_LIBELLE = 1
    };

    ui->setupUi(this);

 // Connexion classique au SGBD.
```

Lier le modèle à un QTableView

```
    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
    db.setDatabaseName("monDSN");
    db.setPassword("plop");
    db.setUserName("alain.defrance");

    model = new QSqlTableModel(this);
    model->setTable("Rubrique");
    model->setSort(RUBRIQUE_LIBELLE, Qt::AscendingOrder); // Définition du critère de tri
    model->setHeaderData(RUBRIQUE_LIBELLE, Qt::Horizontal, "Libelle"); // Définition de l'entête
    model->select();

    ui->tableAvecModelRubrique->setModel(model); // Liaison entre le modèle et le tableau

 ui->tableAvecModelRubrique->setColumnHidden(RUBRIQUE_ID, true); // On masque la colonne des identifiants

 ui->tableAvecModelRubrique->resizeColumnsToContents(); // On redimentionne les colonnes en fonction de leurs con

}
```

Qt permet nativement l'édition des données au travers de l'interface graphique sans rien rajouter. Le composant graphique agit directement sur le modèle et le gère sans avoir besoin d'aide.

VI-C - Déléguer les jointures à l'interface graphique

Dans une structure relationnelle nous avons très souvent besoin de gérer les jointures afin de rendre transparente la séparation des différentes entités.
Il est possible que nous souhaitions obtenir la liste des articles avec leurs rubrique associé, et pourquoi pas permettre de les changer de rubrique. Ceci est extrêmement simple à faire avec Qt et nous allons voir comment.
Tout d'abord, nous allons utiliser des **QSqlRelationalTableModel** qui jouent exactement le même rôle que **QSqlTableModel**, mis à part qu'ils permettent de définir des jointures. Nous appellerons la méthode **QSqlRelationalTableModel::setRelation(int, const QSqlRelation &)**. Le type QSqlRelation permet de définir le critère de jointure.

Modification avec un modèle

```
win_Form::win_Form(QWidget *parent)
```

Modification avec un modèle

```
    : QWidget(parent), ui(new Ui::win_FormClass)
{
 // On défini des constances pour simplifier l'utilisation des indices de colonne.
    enum {
        ARTICLE_ID = 0,
        ARTICLE_TITRE = 1,
        ARTICLE_AUTEUR = 2,
        ARTICLE_RUBRIQUE = 3
    };

    ui->setupUi(this);

 // Connexion classique au SGBD.
    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
    db.setDatabaseName("monDSN");
    db.setPassword("plop");
    db.setUserName("alain.defrance");

    modelArticle = new QSqlRelationalTableModel(this);
    modelArticle->setTable("Article");

  // On lie vers la table Rubrique avec la colonne id_rubrique, et on affichera libelle_rubrique au lieu de id_ru

 modelArticle->setRelation(ARTICLE_RUBRIQUE, QSqlRelation("Rubrique", "id_rubrique", "libelle_rubrique"));
    modelArticle->setSort(ARTICLE_TITRE, Qt::AscendingOrder);
    modelArticle->setHeaderData(ARTICLE_RUBRIQUE, Qt::Horizontal, "Rubrique");
    modelArticle->setHeaderData(ARTICLE_TITRE, Qt::Horizontal, "Titre");
    modelArticle->setHeaderData(ARTICLE_AUTEUR, Qt::Horizontal, "Auteur");
    modelArticle->select();

    ui->tableAvecModelArticle->setModel(modelArticle);
    ui->tableAvecModelArticle->setItemDelegate(new QSqlRelationalDelegate(this));
    ui->tableAvecModelArticle->setColumnHidden(ARTICLE_ID, true);
    ui->tableAvecModelArticle->setSelectionBehavior(QAbstractItemView::SelectRows);
    ui->tableAvecModelArticle->resizeColumnsToContents();

}
```

On voit que Qt s'est chargé de faire la jointure, et gère toujours la mise à jour des données, y compris des données faisant parti des partie des critères de jointure.

VI-D - Les formulaires maître/détail respectant l'intégrité référentielle

C'est maintenant que nous allons aborder le plus intéressant. Dans beaucoup d'applications nous devons gérer des listes, puis des sous listes dépendantes de la première, et devant bien entendu être liées à cette première, c'est-à-dire être mise à jour en fonction de la sélection de la première.
Ici, nous voulons qu'à la sélection de l'article, la liste des chapitres se mette à jour, et que le tout reste éditable. Pour cela nous allons utiliser des modèles comme nous l'avons fait jusqu'à présent. Nous appliquerons un filtre dynamique au modèle gérant les chapitres à la sélection de l'article.

Modification avec un modèle

```
win_Form::win_Form(QWidget *parent)
    : QWidget(parent), ui(new Ui::win_FormClass)
{
 // On défini des constances pour simplifier l'utilisation des indices de colonne.
    enum {
        ARTICLE_ID = 0,
        ARTICLE_TITRE = 1,
        ARTICLE_AUTEUR = 2,
        ARTICLE_RUBRIQUE = 3
    };

    enum {
        CHAPITRE_ID = 0,
        CHAPITRE_TITRE = 1,
        CHAPITRE_ARTICLE = 2
    };

    ui->setupUi(this);

 // Connexion classique au SGBD.
    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
    db.setDatabaseName("monDSN");
    db.setPassword("plop");
    db.setUserName("alain.defrance");

 // Ce que nous avons coder précédement
    modelArticle = new QSqlRelationalTableModel(this);
    modelArticle->setTable("Article");

 modelArticle->setRelation(ARTICLE_RUBRIQUE, QSqlRelation("Rubrique", "id_rubrique", "libelle_rubrique"));
    modelArticle->setSort(ARTICLE_TITRE, Qt::AscendingOrder);
    modelArticle->setHeaderData(ARTICLE_RUBRIQUE, Qt::Horizontal, "Rubrique");
    modelArticle->setHeaderData(ARTICLE_TITRE, Qt::Horizontal, "Titre");
    modelArticle->setHeaderData(ARTICLE_AUTEUR, Qt::Horizontal, "Auteur");
    modelArticle->select();

 // On utilise un nouveau modèle pour les chapitres
    modelChapitre = new QSqlRelationalTableModel(this);
    modelChapitre->setTable("Chapitre");
    modelChapitre->setSort(ARTICLE_TITRE, Qt::AscendingOrder);
    modelChapitre->setHeaderData(CHAPITRE_TITRE, Qt::Horizontal, "Titre du chapitre");
    modelChapitre->select();

 // On applique le modèle d'article au tableau qui lui est réservé
    ui->tableAvecModelArticle->setModel(modelArticle);
    ui->tableAvecModelArticle->setItemDelegate(new QSqlRelationalDelegate(this));
    ui->tableAvecModelArticle->setColumnHidden(ARTICLE_ID, true);
    ui->tableAvecModelArticle->setSelectionBehavior(QAbstractItemView::SelectRows);
    ui->tableAvecModelArticle->resizeColumnsToContents();

 // Pareil pour les chapitres
    ui->tableAvecModelChapitre->setModel(modelChapitre);
    ui->tableAvecModelChapitre->setItemDelegate(new QSqlRelationalDelegate(this));
    ui->tableAvecModelChapitre->setColumnHidden(CHAPITRE_ID, true);
    ui->tableAvecModelChapitre->setSelectionBehavior(QAbstractItemView::SelectRows);
    ui->tableAvecModelChapitre->resizeColumnsToContents();
```

Modification avec un modèle

```
  // On lie l'evenement de selection de la liste d'article a un slot générant le filtre dynamiquement
    connect(ui->tableAvecModelArticle->selectionModel(),
            SIGNAL(currentRowChanged(const QModelIndex &,
                                     const QModelIndex &)),
            this, SLOT(changementArticle(const QModelIndex &)));


}

// Création et attribution du nouveau filtre
void win_Form::changementArticle(const QModelIndex &index)
{
    if(index.isValid())
    {
        QSqlRecord record = modelArticle->record(index.row());
        int id = record.value("id_article").toInt();
        modelChapitre->setFilter(QString("id_article = %1").arg(id));
    }
}
```

Une fois de plus, en quelques lignes de code, nous arrivons à nos fins. Ici, tout est éditable et les contraintes d'intégrité resteront préservées.

VII - Conclusion

Après un rapide tour des possibilités qu'offre Qt en terme d'accès aux données, nous avons de quoi gérer efficacement et rapidement la liaison entre un SGBD et l'interface graphique. Qt permet en quelques lignes de codes de lier des données à l'interface graphique, ce qui permet de gagner un temps colossal dans le développement. Nous aurions dû gérer le parcours , la mise à jour, et bien sûr faire attention à l'intégrité des données (contraintes d'intégrité référentielle), ici Qt se charge de tout.

VIII - Remerciements

Merci à **Alp**, **fsmrel**, **dourouc05**, **ced**, **IrmatDen**, **Vincent Rogier** pour leurs relectures et idées.