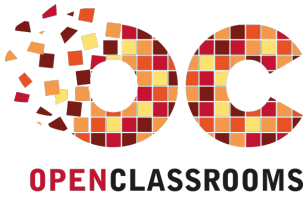
Maîtrisez les bases de données avec QtSQL

Par Dadouchi A.
et Perkele



www.openclassrooms.com

Dernière mise à jour le 5/08/2010

Sommaire

Maîtrisez les bases de données avec QtSQL

Par  Perkele et Dadouchi A.

Mise à jour : 05/08/2010
Difficulté : Facile Durée d'étude : 2 heures

Dans ce tutoriel, nous allons voir ensemble l'utilisation des bases de données dans un projet Qt.

Pour suivre ce tutoriel, il est donc nécessaire d'avoir des bases en C++, de connaître un minimum le framework Qt, de savoir utiliser un S.G.B.D., et d'avoir des bases en SQL.
Si ce n'est pas le cas, vous pouvez lire le tutoriel de C++ en totalité, ainsi que la partie 3 du tutoriel de PHP qui sont présents sur ce site.
Sommaire du tutoriel :

Préparation avant codage

Obtenir une Base de Données

Pour commencer à utiliser une BDD, il faut bien en avoir une, non ?
Si vous en possédez une, vous pouvez passer votre chemin, et aller directement à la partie suivante.

Tout d'abord, si vous voulez juste tester pour voir comment cela marche, et seulement sur votre PC, le plus simple est directement d'en installer une chez vous. Le SGBD que j'utiliserai dans le tutoriel est MySQL, vous pouvez l'installer en le téléchargeant sur le site officiel et en l'installant. Si vous êtes sous GNU/Linux, il se trouve à coup sûr dans vos dépôts.
Attention ! Il vous faut aussi installer la librairie côté client du SGBD choisi !

C'est bon ? vous avez tout ?
C'est parti pour l'aventure !

[Qt] Se connecter, faire des requêtes, et les traiter

Dans cette partie, nous allons apprendre à nous servir de toutes ces magnifiques choses, ne perdons donc pas de temps, allons-y.

Voici donc le code de base que je vous propose pour commencer à étudier le module QtSql.

Code : C++

```
#include <QCoreApplication>
#include <QtSql>
#include <iostream>

#define q2c(string) string.toStdString()

int main(int countArg, char **listArg)
{
    QCoreApplication app(countArg, listArg);
```

```
    std::cout << std::flush;
    return app.exec();
}
```

Voici ce qui est à noter dans le code :

- Peut-être l'avez-vous remarqué avec « QCoreApplication », mais je n'utiliserai pas d'interface graphique pour faire ce code ;
- La macro-définition q2c est simplement faite pour faciliter la conversion de QString vers std::string ; 
- Quand l'on écrit directement avec std::cout, le texte ne sera pas dans la console, pour cela, il faut utiliser std::flush pour forcer l'affichage.

Étant donné que nous n'utilisons pas de GUI (il vous faudrait plus de temps à modifier l'ancien qu'à écrire le nouveau ), ne générez pas votre .pro avec `qmake -project`, utilisez celui-ci :

Code : Autre

```
QT       += sql # On rajoute SQL
QT       -= gui # On enlève la GUI
TARGET = test # La cible de compilation, le résultat
CONFIG   += console # On rajoute la console
TEMPLATE = app # On veut une application, oui
SOURCES += main.cpp # La liste des fichiers de source
```

Et si vous compilez puis exécutez... vous tomberez sur une console vide. Et c'est normal, nous n'avons rien codé, pour l'instant ! 

Connexion à la BDD

Pour commencer, il faut faire la connexion. Pour ce faire, on va créer un objet de la classe QSqlDatabase.
Comme valeur, on va lui assigner le retour de la méthode statique addDatabase() qui reçoit en paramètre sous forme de chaîne de caractères l'une des valeurs du tableau se trouvant dans la documentation.
Dans ce tutoriel, étant donné que je prendrai en exemple une base de données MySQL, nous prendrons la valeur QMYSQL.

Pour résumer ce que je viens de dire, rajoutez ceci dans le code :

Code : C++

```
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
```

Maintenant, nous allons préciser l'hôte, le nom d'utilisateur, le mot de passe, et la base de données que l'on utilisera. Pour ce faire, on utilisera respectivement les méthodes setHostName(), setUserName(), setPassword() et setDatabaseName(), qui prennent chacunes un QString renseignant l'information que l'on modifie.
Pour continuer, nous allons utiliser open(), qui ouvre la connexion, et renvoie un booléen : true si la connexion a réussie, false si elle a échouée. Nous l'utiliserons donc dans une condition.
Pour fermer la base de données, on utilisera la méthode close().

Si je résume, encore une fois, voici notre code :

Code : C++

```
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
db.setHostName("localhost");
db.setUserName("root");
db.setPassword("");
db.setDatabaseName("tuto");
if(db.open())
{
    std::cout << "Vous êtes maintenant connecté à " <<
q2c(db.hostName()) << std::endl;
```

```
        db.close();
    }
    else
    {
        std::cout << "La connexion a échouée, désolé" << std::endl;
    }
```

Vous pouvez donc déjà tester la connexion en utilisant le code ci-dessus.
Ça marche ? Oui ? Parfait. 

Si ce n'est pas le cas, vérifiez que les identifiants sont corrects ou que vous êtes, dans le cas d'un serveur externe, autorisés à accéder à la BDD. Pensez aussi à placer les DLL de l'API de votre SGBD ainsi que ceux de QtSQL.
Remarque : La méthode QSqlDatabase::lastError retourne un objet de type QSqlError, et dedans, il y a une méthode text() qui explique la source du problème !
Vous pouvez donc le connaître ainsi :

Code : C++

```
    //...
    else
    {
        std::cout << "La connexion a échouée, désolé :(" << std::endl <<
    q2c(db.lastError().text()) << std::endl;
    }
```

Effectuer une requête

Tout d'abord, sachez qu'une requête est représentée par la classe QSqlQuery. Elle peut prendre un paramètre au constructeur, ou aucun (ce que je vous conseille, pour n'utiliser qu'un objet de la classe QSqlQuery et ne pas en recréer à chaque requête).

Vous créez un objet QSqlQuery, et pour faire une requête, il vous suffira d'utiliser la méthode exec(), en lui passant la requête en paramètre.
Sachez qu'exec() retourne un booléen : true si la requête a réussie, false si elle a échouée. Si c'est une requête qui n'attend pas de réponse (dans le cas de la requête UPDATE, par exemple), cela suffit. Si par contre elle attend une réponse (dans le cas de SELECT, par exemple), on utilisera cette méthode d'une façon un peu différente que nous verrons dans le paragraphe suivant.
En cas d'erreur de la requête, QSqlQuery::lastError() vous retournera un objet de type QSqlError.
Il vous suffira d'utiliser QSqlError::text() pour l'avoir dans un QString, et pouvoir l'afficher !

Par exemple, imaginez que vous avez une table config, contenant deux champs : type et contenu. *type* serait la configuration que l'on fait (titre, taille, ...) et *contenu* serait la valeur de la configuration ("Mon super programme", "165x253", ...).
Vous voudriez modifier le titre, pour le changer en « Mon programme avec BDD v2.0 ». Voici comment vous devrez faire :

Code : C++

```
    QSqlQuery query;
    if(query.exec("UPDATE `config` SET `contenu`='Mon programme avec BDD
    v2.0' WHERE `type`='titre'"))
    {
        std::cout << "Le titre a bien été changé ! :)" << std::endl;
    }
    else
    {
        std::cout << "Une erreur s'est produite. :(" << std::endl <<
    q2c(query.lastError().text()) << std::endl;
    }
```

Récupérer les données

Donc, comme je viens de le dire, dans le cas d'une requête attendant une réponse, tel que SELECT, on utilisera, combinée à la

méthode exec(), la méthode next() qui retourne la valeur suivante de la requête. Tant qu'il reste des lignes dans la réponse, next() retournera true, et modifiera la ligne actuelle dans notre objet.
Pour accéder à une colonne de la ligne, il vous faudra passer par son ID (le numéro qui donne son ordre dans la liste, le premier étant 0), avec QSqlQuery::value(), vous avez plusieurs solutions :

- Vous savez que la requête ne retournera qu'une colonne, dans ce cas, vous utiliserez le premier ID possible (0) ;
- Vous connaissez le nom du champ, vous allez utiliser QSqlRecord::indexOf() qui prend en paramètre le nom du champ dans un QString ;

Maintenant, je veux afficher les valeurs de tous les champs, et récupérer leurs noms : en passant son ID à la fonction QSqlRecord::fieldname(), on peut récupérer son nom.
Si on veut récupérer la totalité des colonnes, il suffit de faire une boucle « tant que x est plus petit que QSqlRecord::count() ». x sera l'ID de la colonne.

Attends une minute, mon bonhomme ! C'est quoi, cette classe QSqlRecord ? Je dois faire un objet ?

Non, il y a plus simple !
La méthode QSqlQuery::record() retournera l'objet QSqlRecord à utiliser !
Vous pourrez donc faire : `query.record().indexOf("colonne1");`

Par exemple, pour afficher le contenu de notre table « config », cela suffira :

Code : C++

```
QSqlQuery query;
if(query.exec("SELECT * FROM config"))
{
    while(query.next())
    {
        std::cout << " Nouvelle entrée" << std::endl;
        for(int x=0; x < query.record().count(); ++x)
        {
            std::cout << " " << q2c(query.record().fieldName(x)) <<
" = " << q2c(query.value(x)) << std::endl;
        }
    }
}
```

Qui a dit que les BDD avec Qt, c'était dur ? 

Les requêtes préparées

Tout d'abord, qu'est-ce-qu'une requête préparée ? C'est une requête comme une autre, sauf qu'au lieu de mettre directement les valeurs, on va mettre des marqueurs (**?** ou un marqueur nominatif), et les remplacer plus tard.
Ainsi, la requête sera réutilisable pour différentes valeurs.

Pour en créer une, on utilise toujours l'objet QSqlQuery. Sauf que nous n'écrirons pas la requête dans la méthode QSqlQuery::exec(), mais dans la méthode QSqlQuery::prepare().
Pour remplacer un marqueur, il y a deux possibilités :

- C'est un marqueur nominatif, on va utiliser la méthode QSqlQuery::bindValue en mettant le nom du marqueur puis la valeur ;
- C'est un point d'interrogation, on utilisera QSqlQuery::addBindValue() en passant la valeur : il faudra l'utiliser dans l'ordre des **?** !

Si je résume :

Code : C++

```
QSqlQuery q;
```

```
//Marqueurs ?
q.prepare("UPDATE matable SET monchamp=?, monchamp2=?");
q.addBindValue("Coucou");
q.addBindValue(42);

//Marqueurs nominatifs
q.prepare("UPDATE matable SET monchamp=:premierevaleur,
monchamp2=:deuxiemevaleur");
q.bindValue(":premierevaleur", "Coucou");
q.bindValue(":deuxiemevaleur", 42);
```

À noter que les marqueurs nominatifs commencent par le caractère **:** (**Doubles-points**).

Vous avez aussi une deuxième possibilité :
On veut executer cette même requête plusieurs fois. On va utiliser QSqlQuery::addBindValue() pour les deux types de marqueurs, sauf que l'on passera un QVariantList en paramètre.
Attention ! Un QVariantList passé correspond aux valeurs d'un marqueur dans les différentes requêtes, et non aux valeurs des marqueurs dans chacunes des requêtes !
Mais un exemple vaut mieux qu'un avertissement, voici celui de la documentation :

Code : C++

```
QSqlQuery q;
q.prepare("insert into myTable values (?, ?)");

QVariantList ints;
ints << 1 << 2 << 3 << 4;
q.addBindValue(ints);

QVariantList names;
names << "Harald" << "Boris" << "Trond" <<
QVariant(QVariant::String);
q.addBindValue(names);
```

Bonhomme, c'est bien beau tout ça, mais ça m'aide pas à exécuter une requête ! J'utilise exec() tout seul ?

Et bien, c'est presque la solution ! Il va falloir utiliser QSqlQuery::execBatch(). 

Pour compléter le code ci-dessus, voici comment on exécuterai la requête :

Code : C++

```
QSqlQuery q;
q.prepare("insert into myTable values (?, ?)");

QVariantList ints;
ints << 1 << 2 << 3 << 4;
q.addBindValue(ints);

QVariantList names;
names << "Harald" << "Boris" << "Trond" <<
QVariant(QVariant::String);
q.addBindValue(names);

if (q.execBatch())
{
    std::cout << "Ça marche ! :)" << std::endl;
}
else
{
    std::cout << "Ça marche pas ! :(" << std::endl;
}
```

Pour le reste (gestion des erreurs, récupération des données), le fonctionnement est le même.
Il y a tant de choses que j'aurais pu rajouter dans ce tutoriel, tant de choses à apprendre, tant d'astuces à utiliser, tant de choses qui fait que ce tutoriel n'est pas complet.
Par contre, la documentation, elle, est très complète, n'hésitez pas à regarder dedans !

- Documentation de Qt ;
- Documentation du module QtSql ;
- Documentation de la classe QSqlDatabase ;
- Documentation de la classe QSqlQuery ;
- Documentation de la classe QSqlRecord ;
- Documentation de la classe QSqlError.