Etudiant C

HOUDAYER Pierre

BTS2

Dossier de conception détaillée/ réalisation

**Projet 1 :**

**Consignes et assistance vidéolink : Base de données locale**

BTS IRIS 2ème année

Immaculée Conception – LAVAL

Sommaire

[I – La création et la connexion à la base de données. 3](#_Toc421541999)

[I.A – Introduction. 3](#_Toc421542000)

[I.B – Le module QtSql et ses pilotes. 3](#_Toc421542001)

[I.C – Connexion à la base de données. 3](#_Toc421542002)

[I.D – Requêtes sans retour de données. 4](#_Toc421542003)

[I.E – Requête avec retour de données. 5](#_Toc421542004)

[II- Les Interfaces Homme Machine 5](#_Toc421542005)

[II.A – Interface de maintenance 5](#_Toc421542006)

# I – La création et la connexion à la base de données.

## I.A – 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 Système de Gestion de Bases de Données (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, mais nous utiliserons Qt pour sa facilité d'utilisation, sa portabilité et l’intégration dans l’application LC.

Le langage utilisé sera C++ et le module QtSql.

## I.B – Le module QtSql et ses pilotes.

Différents drivers sont utilisables au niveau de la connexion avec le module QtSql, 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.

Les pilotes (drivers) compatible avec le module QtSql sont répertoriés dans le tableau suivant :

|  |  |  |
| --- | --- | --- |
| 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 |

Le pilote que nous utiliserons est naturellement QSQLITE du fait qu’il pilote une base de données avec la librairie SQLite que nous avons précédemment choisi (dernière version 3.8.10.1).

## I.C – Connexion à la base de données.

Avant toute chose nous devons modifier le .pro, avec Qt, il faut précompiler le projet avec qmake pour qu'il puisse être compilé correctement. Il faut modifier ce fichier de projet pour que les classes d'accès aux données soient accessibles. Pour ce faire il suffit d'ajouter ceci au .pro : « QT += sql », On obtient alors l’accès au module QtSql et ses pilotes.

Nous allons maintenant utiliser notre pilote QSQLITE.

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 donc dans notre cas : QSQLITE.

Voici un petit programme simple pour créer une base de données, illustrant ce principe de création et connexion a une base de données :

#**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**("QSQLITE");

db.**setDatabaseName**("LC\_Database.sqlite");

if(!db.**open**())

{

**QMessageBox::critical**(0, **QObject::tr**("Database Error"), db.**lastError**().**text**());

}

window->**show**();

return app.**exec**();

}

## I.D – Requêtes 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, 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 différemment dans le sens où il n'existe qu'un seul moyen d'envoyer la requête. Libre à nous, 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).

Voici le formalisme à utiliser pour envoyer des requêtes au format SQL au moteur de la base de données :

**QSqlQuery** query;

query.**exec**("INSERT INTO Table(Column\_Name\_1, Column\_Name\_2) VALUES ('Value\_ Column\_Name\_1', 'Value\_ Column\_Name\_2')");

## I.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 retour de la requête stocké au sein de l’instance QSqlQuery afin de récupérer et traiter un à un les lignes retournés par la requête.

Pour avoir accès aux données reçues, on utilise la méthode QSqlQuery::value(int) qui nous permet d’identifier la colonne (champ de la table) avec un son numéro en commençant par la gauche que l’on souhaite traiter.

Récupérer ces données nécessite de connaître parfaitement la construction de la table sélectionnée pour identifier les colonnes correctement.

Voici un exemple de l’utilisation de la méthode QSqlQuery::value(int) pour récupérer les données.

**QSqlQuery** query;

query.**exec**("SELECT \* FROM Table");

while(query.next())

{

int Value\_Column\_Name\_1 = query.**value**(0).**toInt**();

**QString** Value\_Column\_Name\_2 = query.**value**(1).**toString**();

}

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.

## I.F – La classe CSQLite\_Local\_DB.

La conception de la classe CSQLite\_Local\_DB est basée sur les principes énoncés précédemment. Les principales méthodes de la classe sont destinées à communiquer directement avec la base de données. Celle-ci sera développé avec l’environnement Qt-creator en C++.

Nous utilisons un outil de génération de documentation automatique nommé « Doxygen ». La totalité de la description de la classe est représenté à l’adresse suivante :

<http://bts2-projet-1.github.io/html>

Menu « Classes » puis « Liste des classes » : CSQLite\_Local\_DB.

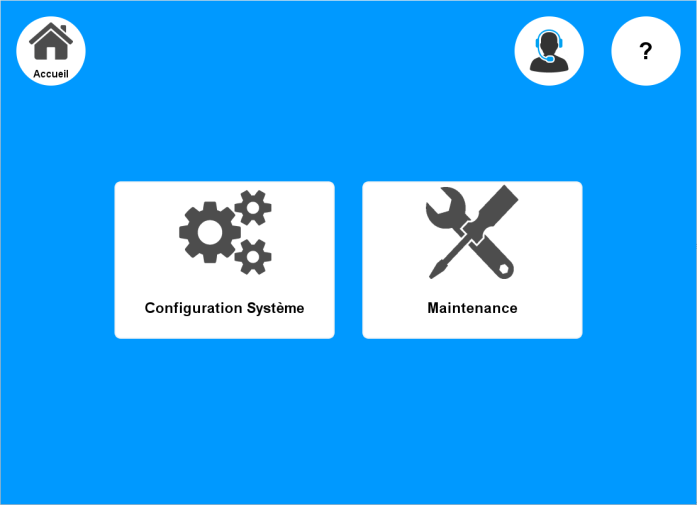
# II- Les Interfaces Homme Machine (IHM).

La conception des IHM sera effectué avec l’environnement de développent Qt-designer qui permet l’édition des interfaces avec un rendu en temps réel.

## II.A – Interface de maintenance.

L’interface de maintenance a pour but de fournir à un opérateur, le moyen rapide de pouvoir gérer le parc de consignes (Ajout, Suppression …). Il sera possible de tester des portes de consignes et la possibilité de pouvoir les activer ou désactiver. Enfin il est possible de pouvoir gérer la base de données en reconstruction par défaut ou d’une ancienne sauvegarde par exemple.

L’interface de maintenance est disponible, après identification, par le bouton Maintenance du menu de configuration.



Disponible

Indisponible



Le menu de maintenance est désormais disponible. Dans l’état actuel du projet, la configuration du site et les redémarrages ne sont pas disponible.

### II.A.1 – Gestion de la base de données.

## II.B – Interface de Configuration du site.