Activité : Base de données

Thierry Vaira <tvaira@free.fr>

2016-2019 (rev. 1.1)

Table des matières

Base de données

Expression du besoin

De nombreuses applications manipulent des données et il est souvent nécessaire des les stocker dans des base de données. Ces base de données peuvent être exploités par des requêtes SQL.

Notions de base

Une base de données (*database*) est un « conteneur » permettant de stocker et de retrouver l'intégralité de données brutes ou d'informations. Dans la très grande majorité des cas, ces informations sont très structurées, et la base est localisée dans un même lieu et sur un même support.

Le dispositif comporte un système de gestion de base de données (SGBD) : un logiciel moteur qui manipule la base de données et dirige l'accès à son contenu. De tels dispositifs comportent également des logiciels applicatifs, et un ensemble de règles relatives à l'accès et l'utilisation des informations.

Une base de données relationnelle est une base de données où l'information est organisée dans des tableaux à deux dimensions appelés des relations ou tables. Les lignes de ces relations sont appelées des *nuplets* (tuples) ou enregistrements. Les noms des colonnes (ou champs) sont appelées des attributs.

Les logiciels qui permettent de créer, utiliser et maintenir des bases de données relationnelles sont des système de gestion de base de données relationnels (SGBDR).

Pratiquement tous les systèmes relationnels utilisent le langage SQL (*Structured Query Language*) pour interroger les bases de données. Ce langage permet permet de rechercher, d'ajouter, de modifier ou de supprimer des données dans les bases de données relationnelles.

Terminologie

Modèle de données Le schéma ou modèle de données est la description de l'organisation des données. Il renseigne sur les caractéristiques de chaque type de donnée et les relations entre les différentes données qui se trouvent dans la base de données. Il existe plusieurs types de modèles de données (relationnel, entité-association, objet, hiérarchique et réseau).

Entité Une entité est un objet, un sujet, une notion en rapport avec le domaine d'activité pour lequel la base de données est utilisée, et concernant celui pour lequel des données sont enregistrées (exemple : des personnes, des produits, des commandes, des réservations, . . .).

Attribut Un attribut est une caractéristique d'une entité susceptible d'être enregistrée dans la base de données. Par exemple une personne (entité), son nom et son adresse (des attributs). Les attributs sont également appelés des champs ou des colonnes.

Enregistrement Un enregistrement est une donnée qui comporte plusieurs champs dans chacun desquels est enregistrée une donnée.

Association Les associations désignent les liens qui existent entre différentes entités, par exemple entre un vendeur, un client et un magasin.

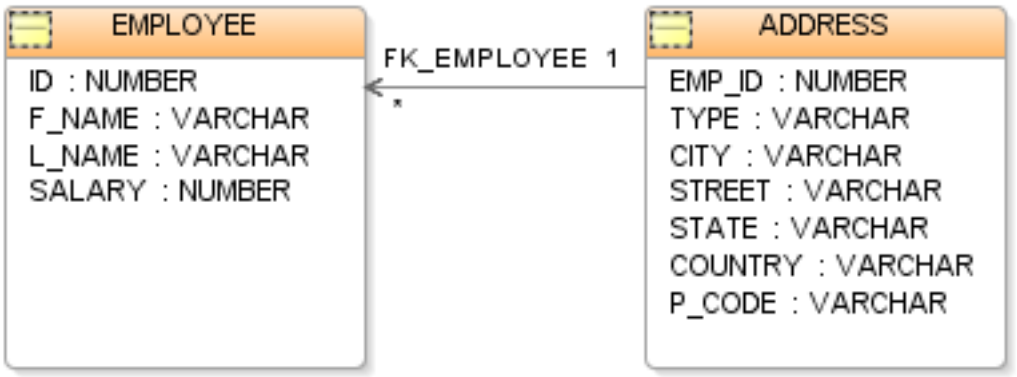
Cardinalité La cardinalité d'une association (d'un lien entre deux entités A et B par exemple) est le nombre de A pour lesquelles il existe un B et inversement. Celle-ci peut être un-a-un, un-a-plusieurs ou plusieurs-à-plusieurs. Par exemple un compte bancaire appartient à un seul client, et un client peut avoir plusieurs comptes bancaires (cardinalité un-a-plusieurs).

Modèle de données relationnel C'est le type de modèle de données le plus couramment utilisé pour la réalisation d'une base de données. Selon ce type de modèle, la base de données est composée d'un ensemble de tables (les relations) dans lesquelles sont placées les données ainsi que les liens. Chaque ligne d'une table est un enregistrement.

Base de données relationnelle C'est une base de données organisée selon un modèle de données de type relationnel, à l'aide d'un SGBD permettant ce type de modèle.

Clé primaire Dans les modèles de données relationnels, la clé primaire est un attribut dont le contenu est différent pour chaque enregistrement de la table, ce qui permet de retrouver un et un seul enregistrement (unicité). Dans les modèles de données relationnels, une clé étrangère est un attribut qui contient une référence à une donnée connexe (la valeur de la clé primaire de la donnée connexe).

SQL (*Structured Query Language*) C'est un langage de requête structurée et normalisé servant à exploiter des bases de données relationnelles. La partie langage de manipulation des données de SQL permet de rechercher, d'ajouter, de modifier ou de supprimer des données dans les bases de données relationnelles.



Voir aussi : Schéma relationnel

Exemples SQL

Recherche des lignes (aussi appelés tuples) dans une table existante :

```
SELECT nom, service
FROM   employe
WHERE  statut = 'stagiaire'
ORDER  BY nom;
```

Insère une ligne (aussi appelés tuple) dans une table existante :

```
INSERT INTO employe (nom, service, statut)
VALUES ('toto', 'rd', 'developpeur');
```

Modifie un tuple existant dans une table :

 © 2019 tv <tvaira@free.fr>

```
UPDATE employe
SET statut = 'chef'
WHERE nom = 'toto';
```

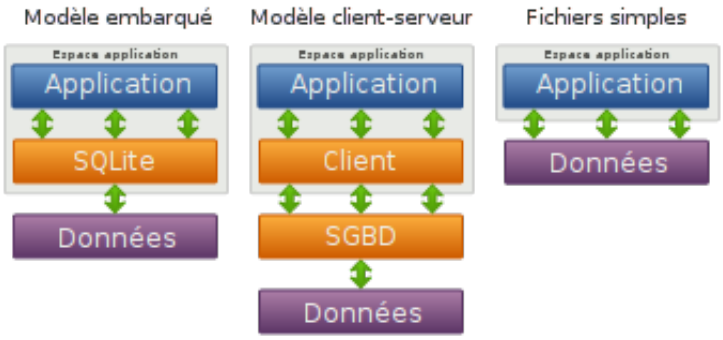
Supprime un tuple existant dans une table :

```
DELETE FROM employe
WHERE nom = 'toto';
```

MySQL vs SQLite

MySQL est un système de gestion de bases de données relationnelles (SGBDR). Il est distribué sous une double licence GPL et propriétaire. Il fait partie des logiciels de gestion de base de données les plus utilisés au monde, autant par le grand public (applications web principalement) que par des professionnels, en concurrence avec Oracle, Informix et Microsoft SQL Server. MySQL est un serveur de bases de données relationnelles SQL. Il est multi-thread et multi-utilisateur. MySQL fonctionne sur de nombreux systèmes d'exploitation différents, incluant Linux, Mac OS X et Windows. Les bases de données sont accessibles en utilisant les langages de programmation C, C++, VB, VB .NET, C#, Delphi/Kylix, Eiffel, Java, Perl, PHP, Python, Windev, Ruby et Tcl. Une API spécifique est disponible pour chacun d'entre eux. MySQL fait partie du quatuor LAMP : Linux, Apache, MySQL, PHP. Il appartient également à ses variantes WAMP (Windows) et MAMP (Mac OS).

SQLite est une bibliothèque écrite en C qui propose un moteur de base de données relationnelle accessible par le langage SQL. Contrairement aux serveurs de bases de données traditionnels, comme MySQL ou PostgreSQL, sa particularité est de ne pas reproduire le schéma habituel client-serveur mais d'être directement intégrée aux programmes. L'intégralité de la base de données (déclarations, tables, index et données) est stockée dans un fichier indépendant de la plateforme. SQLite est le moteur de base de données le plus distribué au monde, grâce à son utilisation dans de nombreux logiciels grand public comme Firefox, Skype, Google Gears, dans certains produits d'Apple, d'Adobe et de McAfee et dans les bibliothèques standards de nombreux langages comme PHP ou Python. De par son extrême légèreté (moins de 300 Kio), il est également très populaire sur les systèmes embarqués, notamment sur la plupart des smartphones modernes : l'iPhone ainsi que les systèmes d'exploitation mobiles Symbian et Android l'utilisent comme base de données embarquée.



Accès distant pour MySQL

Si votre application s'exécute sur la même machine que votre serveur de base de données, vous pourrez indiquer `localhost` comme nom de machine. Sinon, il vous faudra préciser le nom réseau ou l'adresse IP du serveur à joindre et configurer un accès distant pour cette application :

— éditer le fichier le fichier `/etc/mysql/my.cnf` : dans la section `[mysqld]`, indiquer pour le paramétre `bind-address` l'adresse IP de votre interface d'écoute ou la valeur `0.0.0.0` pour toutes les interfaces réseau de votre serveur

 © 2019 tv <tvaira@free.fr>

```
[mysqld]
#
# * Basic Settings
#
user            = mysql
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
port            = 3306
basedir         = /usr
datadir         = /var/lib/mysql
tmpdir          = /tmp
lc-messages-dir = /usr/share/mysql
skip-external-locking
#
# Instead of skip-networking the default is now to listen only on
# localhost which is more compatible and is not less secure.
#bind-address           = 127.0.0.1
#bind-address           = 0.0.0.0
bind-address            = 192.168.52.123
```

— ajouter un utilisateur (ici **rpi**) pour lequel vous autorisez l'accès distant (%) à cette base de données :

Utilisateurs ayant accès à "dmi"

Utilisateur	Client	Type	Privilèges	«Grant »	Action
debian-sys-maint	localhost	global	ALL PRIVILEGES	Oui	Changer les privilèges
root	127.0.0.1	global	ALL PRIVILEGES	Oui	Changer les privilèges
root	::1	global	ALL PRIVILEGES	Oui	Changer les privilèges
root	iris-hp-pro-3500-series	global	ALL PRIVILEGES	Oui	Changer les privilèges
root	localhost	global	ALL PRIVILEGES	Oui	Changer les privilèges
rpi	%	spécifique à cette base de données	ALL PRIVILEGES	Non	Changer les privilèges

— redémarrer le service MySQL :

```
/etc/init.d/mysql restart
```

Remarque : en cas de problème, vérifier si un pare-feu (*firewall*) est actif et si les connexions entrantes sur le port 3306 (port d'écoute par défaut du serveur MySQL) ne sont pas bloquées

API Qt

Qt fournit de nombreuses classes pour la gestion des base de données. Il faudra activer le module dans son fichier de projet `.pro` pour pouvoir accéder aux classes :

```
...
QT += sql
...
```

Remarque : Il faudra aussi disposer d'un pilote de base de données comme pour MySQL (QMYSQL) ou SQLite (QSQLITE2 ou QSQLITE).

 © 2019 tv <tvaira@free.fr>

On utilisera alors la classe `QSqlDatabase` qui permet la connexion à une base de données.

- La classe QSqlDatabase Qt4 (en)
- La classe QSqlDatabase Qt5 (en)
- QSqlDatabase (fr)

Et ensuite la classe `QSqlQuery` pour exécuter des requêtes SQL :

- La classe QSqlQuery Qt4 (en)
- La classe QSqlQuery Qt5 (en)

Objectifs

Être capable d'exécuter des requêtes simples sur une base de données MySQL sous Qt.

Préparation

- Pour **Qt 4** :

Pour utiliser **MySQL** avec Qt, il vous faudra tout d'abord installer `libqt4-sql-mysql` :

```
$ sudo apt-get install libqt4-sql-mysql
```

Pour utiliser **SQLite** avec Qt, il vous faudra tout d'abord installer `libqt4-sql-sqlite` :

```
$ sudo apt-get install libqt4-sql-sqlite
```

- Pour **Qt 5** :

Pour utiliser **MySQL** avec Qt, il vous faudra tout d'abord installer `libqt5sql5-mysql` :

```
$ sudo apt-get install libqt5sql5-mysql
```

Pour utiliser **SQLite** avec Qt, il vous faudra tout d'abord installer `libqt5sql5-sqlite` :

```
$ sudo apt-get install libqt5sql5-sqlite
```

Pour utiliser la base de données d'exemple (ici on utilisera la base de données `test` qui existe par défaut), on vous fournit un fichier SQL à déployer sur votre serveur MySQL :

```
USE test;

CREATE TABLE `mesures` (
  `id` int(10) NOT NULL AUTO_INCREMENT,
  `date` date NOT NULL,
  `heure` time NOT NULL,
  `temperature` float NOT NULL,
  PRIMARY KEY (`id`)
);

INSERT INTO `mesures` (`id`, `date`, `heure`, `temperature`) VALUES
(1, '2009-09-08', '08:00:00', 35.23),
(2, '2009-09-08', '08:01:00', 35.1),
(3, '2009-09-08', '08:02:00', 34.45),
(4, '2009-09-08', '08:03:00', 35.02),
(5, '2009-09-08', '08:04:00', 35.53),
(6, '2009-09-09', '08:00:00', 35.23),
(7, '2009-09-09', '08:01:00', 35.1),
(8, '2009-09-09', '08:02:00', 34.45),
(9, '2009-09-09', '08:03:00', 35.02),
(10, '2009-09-09', '08:04:00', 35.53),
(11, '2009-09-09', '08:05:00', 35.12);
```

 © 2019 tv <tvaira@free.fr>

Séquence 1 : utilisation des classes QSqlDatabase et QSqlQuery

Pour manipuler la table `mesures` dans le programme :

```
#include <QtGui>
#include <QDebug>
#include <QtSql/QtSql>
#include <iostream>

int main(int argc,char **argv)
{
   QApplication app(argc,argv);

   QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL"); // ou mettre QSQLITE pour SQLite

   db.setHostName("localhost");
   db.setUserName("root");
   db.setPassword("password");
   db.setDatabaseName("test"); // ou mettre le nom du fichier sqlite

   if(db.open())
   {
       std::cout << "Connexion réussie à " << db.hostName().toStdString() << std::endl;
   }
   else
   {
       std::cout << "La connexion a échouée !" << std::endl;
       return -1;
   }

   // Exemple 1 :
   QSqlQuery query("SELECT temperature FROM mesures");
   while (query.next())
   {
      QString temperature = query.value(0).toString();
      qDebug() << "temperature : " << temperature;
   }

   // Exemple 2 :
   bool retour;
   QString requete = "SELECT * FROM mesures";
   retour = query.exec(requete);
   if(retour)
   {
      qDebug() << "nb enregistrements : " << query.size();
      qDebug() << "nb colonnes : " << query.record().count();
      int fieldNo = query.record().indexOf("temperature");
      while (query.next())
      {
         QString date = query.value(fieldNo).toString();
         qDebug() << "temperature : " << date;
      }
   }
   else  qDebug() << query.lastError().text();

   // Exemple 3 :
   requete = "SELECT * FROM mesures";
   retour = query.exec(requete);
   if(retour)
   {
      while ( query.next() )
      {
         qDebug() << "enregistrement -> ";
         for(int i=0;i<query.record().count();i++)
            qDebug() << query.value(i).toString();
      }
   }
```

 © 2019 tv <tvaira@free.fr>

```
    else  qDebug() << query.lastError().text();

    // Exemple 4 :
    requete = "SELECT count(temperature) AS nb FROM mesures";

    query.exec(requete);
    query.first();
    int nb = query.value(0).toInt();
    qDebug() << "nb mesures : " << nb;

    // Exemple 5 :
    QSqlQuery r;
    // Utilisation des marqueurs '?'
    // INSERT INTO `mesures` (`id`, `date`, `heure`, `temperature`) VALUES (...)
    r.prepare("INSERT INTO mesures (id, date, heure, temperature) VALUES ('', ?, ?, ?)");
    // id en auto-incrément
    r.addBindValue("2009-09-10");
    r.addBindValue("09:01:00");
    r.addBindValue(35.12);
    if (r.exec())
    {
        std::cout << "Insertion réussie" << std::endl;
    }
    else
    {
        std::cout << "Echec insertion !" << std::endl;
        qDebug() << r.lastError().text();
    }

    // Utilisation des marqueurs nominatifs
    r.prepare("INSERT INTO mesures (id, date, heure, temperature) VALUES (:id, :date, :heure, :temperature)");
    r.bindValue(":id", ""); // auto-incrément
    r.bindValue(":date", "2009-09-10");
    r.bindValue(":heure", "09:01:00");
    r.bindValue(":temperature", 34.92);
    if (r.exec())
    {
        std::cout << "Insertion réussie" << std::endl;
    }
    else
    {
        std::cout << "Echec insertion !" << std::endl;
        qDebug() << r.lastError().text();
    }

    db.close();
    return 0;
}
```

Code source : test-mo-sql.zip

Séquence 2 : mise en oeuvre d'une classe BaseDeDonnees

Étant donné que nous possédons qu'une seule base de données, on désire disposer d'un objet unique modélisant notre base de données. Pour cela, on va implémenter le *design pattern* singleton sur notre classe BaseDeDonnees.

```
Un design pattern (ou motif de conception) est un document qui décrit une solution générale à un problème connu
    et récurrent. Le Singleton vise à assurer qu'il n'y a toujours qu'une seule instance d'un objet en
    fournissant une interface pour la manipuler. C'est un des patrons les plus simples. La classe fournit une m
    éthode statique pour obtenir cette unique instance (getInstance() par exemple) et un mécanisme pour empê
    cher la création d'autres instances (en placant le constructeur en private).
```

D'autre part notre application étant décomposé en plusieurs *threads*, il faudra s'assurer que la base de données soit en exclusion mutuelle. Pour cela, on ajoutera un mutex pour garantir l'accès unique à cette ressource critique.

 © 2019 tv <tvaira@free.fr>

```
Une ressource est en exclusion mutuelle si seul un thread peut utiliser la ressource à un instant donné. Une
    section critique une partie de code telle que deux threads ne peuvent s'y trouver au même instant. Un mutex
     est un objet d'exclusion mutuelle (MUTual EXclusion), et est très pratique pour protéger des données
    partagées de modifications concurrentes et pour implémenter des sections critiques. Qt fournit une classe
    QMutexLocker qui permet facilement de gérer une section critique basée sur un QMutex.
```

Toutes les requêtes SQL seront typées en `QString` ainsi que les données retournées.

Pour accèder à la base de données, il vous faut une instance (un objet) puis la connecter :

```
BaseDeDonnees *bdd = BaseDeDonnees::getInstance();

bdd->connecter();
```

Lorsque vous n'avez plus besoin de l'instance sur la base de données, il vous faut la libérer :

```
BaseDeDonnees::detruireInstance();
```

Pour manipuler des requêtes SQL, on distinguera les deux méthodes suivantes :

- `executer()` pour `UPDATE`, `INSERT` et `DELETE` et
- `recuperer()` pour `SELECT`.

Le nombre de données retournées pour une requête `SELECT` pouvant être différent, la méthode `recuperer()` a été surchargée pour traiter éventuellement les situations suivantes :

- un seul champ d'un seul enregistrement : le champ est stocké dans un `QString`

```
QString donnee;
QString requete = "SELECT niveauEauMin FROM Seuils";
bdd->recuperer(requete, donnee);
```

phMin	phMax	temperatureMin	temperatureMax	niveauEauMin
5	9	23	27	40

QString

- plusieurs champs d'un seul enregistrement : les différents champs sont stockés dans un `QStringList`

```
QStringList donnees;
QString requete = "SELECT * FROM Seuils";
bdd->recuperer(requete, donnees);
```

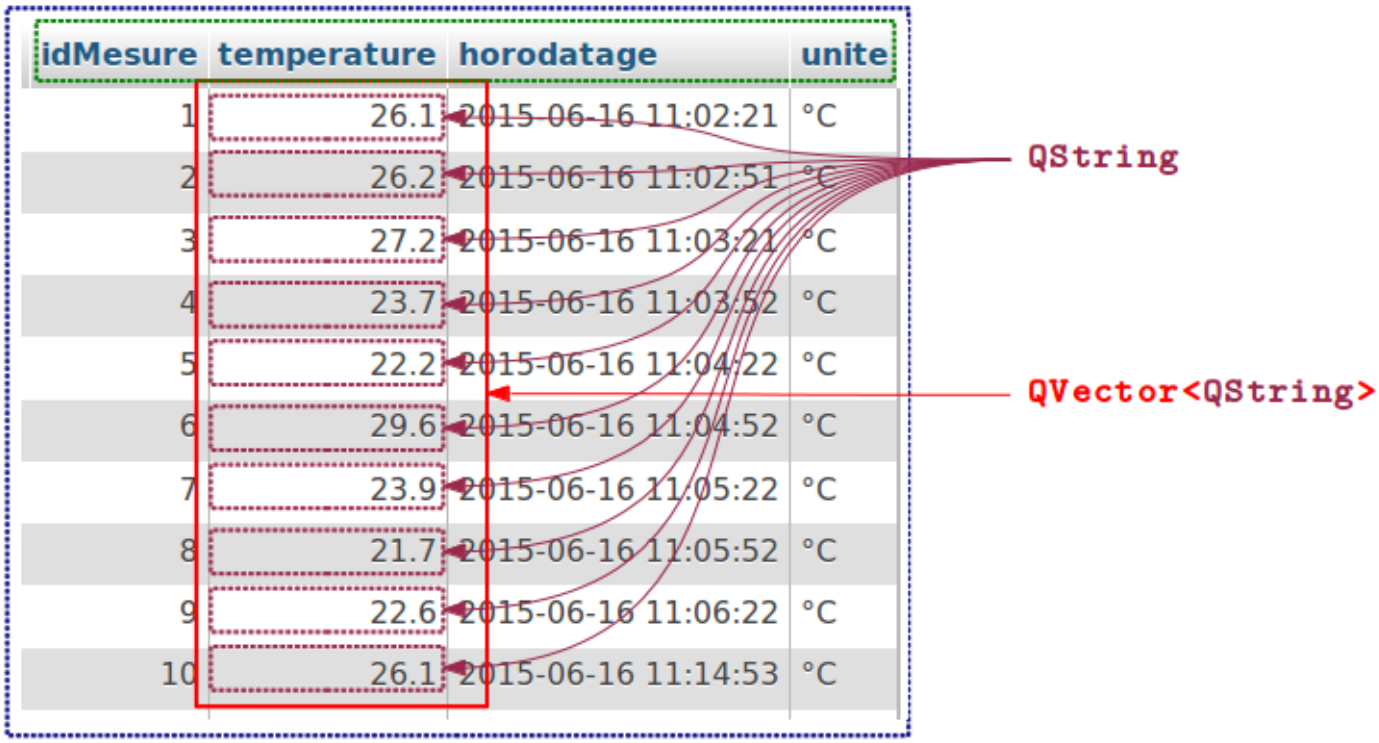
phMin	phMax	temperatureMin	temperatureMax	niveauEauMin
5	9	23	27	40

QStringList

- un seul champ de plusieurs enregistrements : le champ des différents enregistrements est stocké dans un `QVector` de `QString`

 © 2019 tv <tvaira@free.fr>

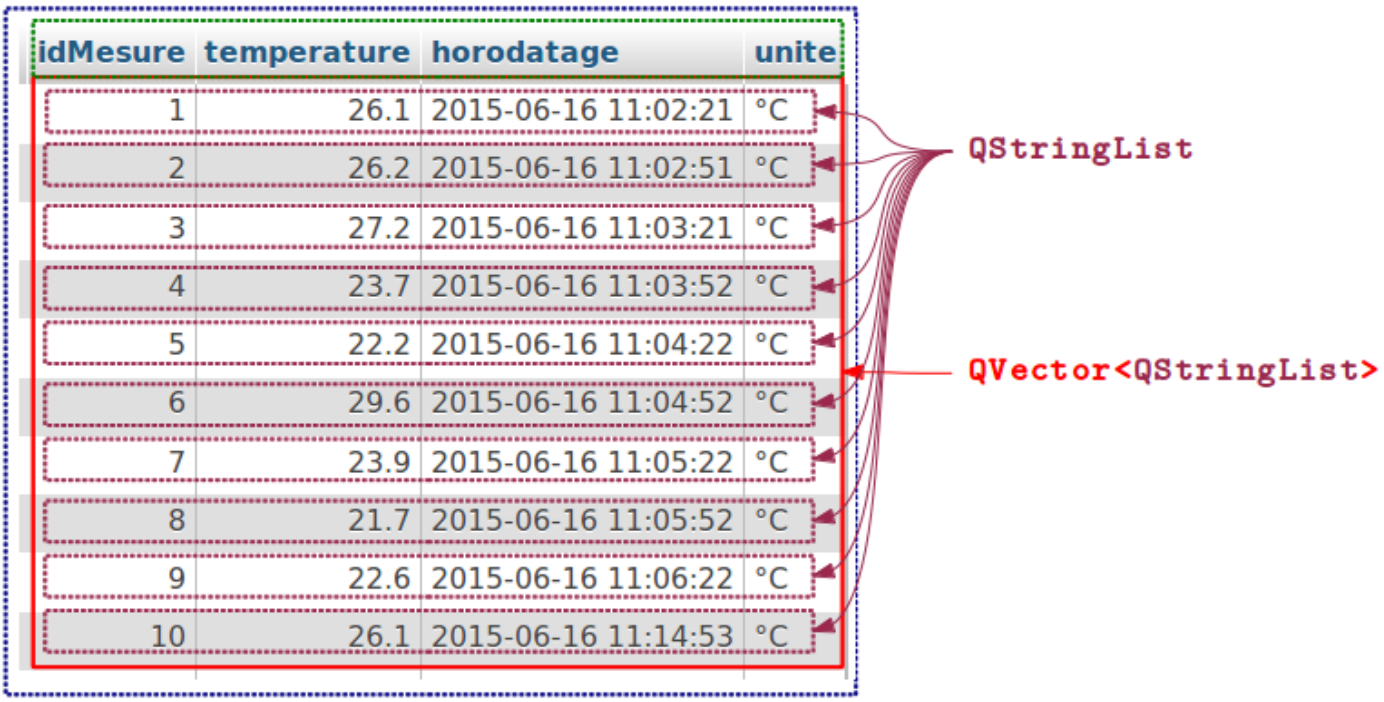
```
QVector<QString> donnees;
QString requete = "SELECT temperature FROM Mesures";
bdd->recuperer(requete, donnees);
```



— plusieurs champs de plusieurs enregistrements : les différents champs des différents enregistrements sont stockés dans un `QVector` de `QStringList`

 © 2019 tv <tvaira@free.fr>

```
QVector<QStringList> donnees;
QString requete = "SELECT * FROM Mesures";
bdd->recuperer(requete, donnees);
```



La déclaration de la classe `BaseDeDonnees` :

```
#ifndef BASEDEDONNEES_H
#define BASEDEDONNEES_H

#include <QObject>
#include <QtSql/QtSql>
#include <QSqlDatabase>
#include <QMutex>

#define DEBUG_BASEDEDONNEES

#define HOSTNAME        "localhost"
#define USERNAME        "root"
#define PASSWORD        "password"
#define DATABASENAME    "test"

class BaseDeDonnees : public QObject
{
    Q_OBJECT
    public:
        static BaseDeDonnees* getInstance();
        static void detruireInstance();

        bool connecter(QString nomBase = DATABASENAME);
        /* uniquement pour : UPDATE, INSERT et DELETE */
        bool executer(QString requete);
        /* uniquement pour : SELECT */
        bool recuperer(QString requete, QString &donnees); // 1 -> 1
        bool recuperer(QString requete, QStringList &donnees); // 1 -> 1..*
        bool recuperer(QString requete, QVector<QString> &donnees); // 1..* -> 1
        bool recuperer(QString requete, QVector<QStringList> &donnees); // 1..* -> 1..*
```

 © 2019 tv <tvaira@free.fr>

```
    private:
        BaseDeDonnees();
        ~BaseDeDonnees();
        static BaseDeDonnees*  baseDeDonnees;
        static int             nbAcces;
        QSqlDatabase           db;
        QMutex                 mutex;
};

#endif // BASEDEDONNEES_H
```

La définition de la classe `BaseDeDonnees` :

```
#include "basededonnees.h"

#include <QDebug>
#include <QMessageBox>

BaseDeDonnees* BaseDeDonnees::baseDeDonnees = NULL;
int BaseDeDonnees::nbAcces = 0;

BaseDeDonnees::BaseDeDonnees()
{
    #ifdef DEBUG_BASEDEDONNEES
    qDebug() << "<BaseDeDonnees::BaseDeDonnees()>";
    #endif
    db = QSqlDatabase::addDatabase("QMYSQL");
}

BaseDeDonnees::~BaseDeDonnees()
{
    #ifdef DEBUG_BASEDEDONNEES
    qDebug() << "<BaseDeDonnees::~BaseDeDonnees()>";
    #endif
}

BaseDeDonnees* BaseDeDonnees::getInstance()
{
    if(baseDeDonnees == NULL)
        baseDeDonnees = new BaseDeDonnees();

    nbAcces++;
    #ifdef DEBUG_BASEDEDONNEES
    qDebug() << "<BaseDeDonnees::getInstance()> nbAcces = " << nbAcces;
    #endif

    return baseDeDonnees;
}

void BaseDeDonnees::detruireInstance()
{
    // instance ?
    if(baseDeDonnees != NULL)
    {
        nbAcces--;
        #ifdef DEBUG_BASEDEDONNEES
        qDebug() << "<BaseDeDonnees::detruireInstance()> nbAcces restants = " << nbAcces;
        #endif
        // dernier ?
        if(nbAcces == 0)
            delete baseDeDonnees;
    }
}

bool BaseDeDonnees::connecter(QString nomBase /*= DATABASENAME*/)
{
```

```
    QMutexLocker verrou(&mutex);
    if(!db.isOpen())
    {
       db.setHostName(HOSTNAME);
       db.setUserName(USERNAME);
       db.setPassword(PASSWORD);
       db.setDatabaseName(nomBase);

       #ifdef DEBUG_BASEDEDONNEES
       qDebug() << "HostName : " << db.hostName();
       qDebug() << "UserName : " << db.userName();
       qDebug() << "DatabaseName : " << db.databaseName();
       #endif
       if(db.open())
       {
           #ifdef DEBUG_BASEDEDONNEES
           qDebug() << QString::fromUtf8("<BaseDeDonnees::connecter()> connexion réussie à %1").arg(db.hostName
    ());
           #endif

           return true;
       }
       else
       {
           qDebug() << QString::fromUtf8("<BaseDeDonnees::connecter()> erreur : impossible de se connecter à la
    base de données !");

           QMessageBox::critical(0, QString::fromUtf8("Test MO"), QString::fromUtf8("Impossible de se connecter
    à la base de données !"));

           return false;
       }
    }
    else
        return true;
}

/* pour : UPDATE, INSERT et DELETE */
bool BaseDeDonnees::executer(QString requete)
{
    QMutexLocker verrou(&mutex);
    QSqlQuery r;
    bool retour;

    if(db.isOpen())
    {
        retour = r.exec(requete);
        #ifdef DEBUG_BASEDEDONNEES
        qDebug() << QString::fromUtf8("<BaseDeDonnees::executer()> retour %1 pour la requete : %2").arg(QString
    ::number(retour)).arg(requete);
        #endif
        if(retour)
        {
            return true;
        }
        else
        {
            qDebug() << QString::fromUtf8("<BaseDeDonnees::executer()> erreur : %1 pour la requête %2").arg(r.
    lastError().text()).arg(requete);

            return false;
        }
    }
    else
        return false;
```

 © 2019 tv <tvaira@free.fr>

```
}

/* uniquement pour récupérer (SELECT) UN champ d'UN seul enregistrement
   Remarque : le champ est stocké dans un QString
 */
bool BaseDeDonnees::recuperer(QString requete, QString &donnees)
{
    QMutexLocker verrou(&mutex);
    QSqlQuery r;
    bool retour;

    if(db.isOpen())
    {
        retour = r.exec(requete);
        #ifdef DEBUG_BASEDEDONNEES
        qDebug() << QString::fromUtf8("<BaseDeDonnees::recuperer(QString, QString)> retour %1 pour la requete :
    %2").arg(QString::number(retour)).arg(requete);
        #endif
        if(retour)
        {
            // on se positionne sur l'enregistrement
            r.first();

            // on vérifie l'état de l'enregistrement retourné
            if(!r.isValid())
            {
                #ifdef DEBUG_BASEDEDONNEES
                qDebug() << QString::fromUtf8("<BaseDeDonnees::recuperer(QString, QString)> résultat non valide
    !");
                #endif
                return false;
            }

            // on récupère sous forme de QString la valeur du champ
            if(r.isNull(0))
            {
                #ifdef DEBUG_BASEDEDONNEES
                qDebug() << QString::fromUtf8("<BaseDeDonnees::recuperer(QString, QString)> résultat vide !");
                #endif
                return false;
            }
            donnees = r.value(0).toString();
            #ifdef DEBUG_BASEDEDONNEES
            qDebug() << "<BaseDeDonnees::recuperer(QString, QString)> enregistrement -> " << donnees;
            #endif
            return true;
        }
        else
        {
            qDebug() << QString::fromUtf8("<BaseDeDonnees::recuperer(QString, QString)> erreur : %1 pour la requ
    ête %2").arg(r.lastError().text()).arg(requete);

            return false;
        }
    }
    else
        return false;
}

/* uniquement pour récupérer (SELECT) plusieurs champs d'UN seul enregistrement
   Remarque : les différents champs sont stockés dans un QStringList
 */
bool BaseDeDonnees::recuperer(QString requete, QStringList &donnees)
{
    QMutexLocker verrou(&mutex);
    QSqlQuery r;
```

```
    bool retour;

    if(db.isOpen())
    {
        retour = r.exec(requete);
        #ifdef DEBUG_BASEDEDONNEES
        qDebug() << QString::fromUtf8("<BaseDeDonnees::recuperer(QString, QStringList)> retour %1 pour la
    requete : %2").arg(QString::number(retour)).arg(requete);
        #endif
        if(retour)
        {
            // on se positionne sur l'enregistrement
            r.first();

            // on vérifie l'état de l'enregistrement retourné
            if(!r.isValid())
            {
                #ifdef DEBUG_BASEDEDONNEES
                qDebug() << QString::fromUtf8("<BaseDeDonnees::recuperer(QString, QStringList)> résultat non
    valide !");
                #endif
                return false;
            }

            // on récupère sous forme de QString la valeur de tous les champs sélectionnés
            // et on les stocke dans une liste de QString
            for(int i=0;i<r.record().count();i++)
                if(!r.isNull(i))
                    donnees << r.value(i).toString();
            #ifdef DEBUG_BASEDEDONNEES
            qDebug() << "<BaseDeDonnees::recuperer(QString, QStringList)> enregistrement -> " << donnees;
            #endif
            return true;
        }
        else
        {
            qDebug() << QString::fromUtf8("<BaseDeDonnees::recuperer(QString, QStringList)> erreur : %1 pour la
    requête %2").arg(r.lastError().text()).arg(requete);
            return false;
        }
    }
    else
        return false;
}

/* uniquement pour récupérer (SELECT) un seul champ de plusieurs enregistrements
   Remarque : le champ des différents enregistrements est stocké dans un QVector de QString
 */
bool BaseDeDonnees::recuperer(QString requete, QVector<QString> &donnees)
{
    QMutexLocker verrou(&mutex);
    QSqlQuery r;
    bool retour;
    QString data;

    if(db.isOpen())
    {
        retour = r.exec(requete);
        #ifdef DEBUG_BASEDEDONNEES
        qDebug() << QString::fromUtf8("<BaseDeDonnees::recuperer(QString, QVector<QString>)> retour %1 pour la
    requete : %2").arg(QString::number(retour)).arg(requete);
        #endif
        if(retour)
        {
            // pour chaque enregistrement
            while ( r.next() )
```

```
            {
                // on récupère sous forme de QString la valeur du champs sélectionné
                data = r.value(0).toString();

                #ifdef DEBUG_BASEDEDONNEES
                //qDebug() << "<BaseDeDonnees::recuperer(QString, QVector<QString>)> enregistrement -> " << data
    ;
                #endif

                // on stocke l'enregistrement dans le QVector
                donnees.push_back(data);
            }
            #ifdef DEBUG_BASEDEDONNEES
            qDebug() << "<BaseDeDonnees::recuperer(QString, QVector<QString>)> enregistrement -> " << donnees;
            #endif
            return true;
        }
        else
        {
            qDebug() << QString::fromUtf8("<BaseDeDonnees::recuperer(QString, QVector<QString>)> erreur : %1
    pour la requête %2").arg(r.lastError().text()).arg(requete);

            return false;
        }
    }
    else
        return false;
}

/* uniquement pour récupérer (SELECT) plusieurs champs de plusieurs enregistrements
   Remarque : les différents champs des différents enregistrements sont stockés dans un QVector de QStringList
 */
bool BaseDeDonnees::recuperer(QString requete, QVector<QStringList> &donnees)
{
    QMutexLocker verrou(&mutex);
    QSqlQuery r;
    bool retour;
    QStringList data;

    if(db.isOpen())
    {
        retour = r.exec(requete);
        #ifdef DEBUG_BASEDEDONNEES
        qDebug() << QString::fromUtf8("<BaseDeDonnees::recuperer(QString, QVector<QStringList>)> retour %1 pour
    la requete : %2").arg(QString::number(retour)).arg(requete);
        #endif
        if(retour)
        {
            // pour chaque enregistrement
            while ( r.next() )
            {
                // on récupère sous forme de QString la valeur de tous les champs sélectionnés
                // et on les stocke dans une liste de QString
                for(int i=0;i<r.record().count();i++)
                    data << r.value(i).toString();

                #ifdef DEBUG_BASEDEDONNEES
                //qDebug() << "<BaseDeDonnees::recuperer(QString, QVector<QStringList>)> enregistrement -> " <<
    data;
                /*for(int i=0;i<r.record().count();i++)
                    qDebug() << r.value(i).toString();*/
                #endif

                // on stocke l'enregistrement dans le QVector
                donnees.push_back(data);
```

 © 2019 tv <tvaira@free.fr>

```
                // on efface la liste de QString pour le prochain enregistrement
                data.clear();
            }
            #ifdef DEBUG_BASEDEDONNEES
            qDebug() << "<BaseDeDonnees::recuperer(QString, QVector<QStringList>)> enregistrement -> " <<
    donnees;
            #endif
            return true;
        }
        else
        {
            qDebug() << QString::fromUtf8("<BaseDeDonnees::recuperer(QString, QVector<QStringList>)> erreur : %1
     pour la requête %2").arg(r.lastError().text()).arg(requete);

            return false;
        }
    }
    else
        return false;
}
```

Un programme de test de la classe `BaseDeDonnees` :

```
#include "basededonnees.h"

int main()
{
    BaseDeDonnees *bdd = BaseDeDonnees::getInstance();

    bdd->connecter();

    bool retour;
    QString requete;
    QString periode;
    QStringList seuils;
    QVector<QStringList> listeAlarmes;
    QStringList alarme;

    // Exemple : récupère un champ d'un seul enregistrement
    // le champ lu est récupéré dans un QString

    // récupère la périodicité de la tâche acquisition des mesures
    requete = "SELECT periode FROM parametres";
    qDebug() << QString::fromUtf8("requête : ") << requete;

    retour = bdd->recuperer(requete, periode);
    if(retour != false)
    {
        qDebug() << QString::fromUtf8("période : ") << periode;
    }
    else
    {
        qDebug() << QString::fromUtf8("erreur !");
    }

    // Exemple : récupère au moins deux champs d'un seul enregistrement
    // les champs lus sont récupérés dans un QStringList

    // récupère les seuils pour la gestion des alarmes
    requete = "SELECT min, max FROM seuils";
    qDebug() << QString::fromUtf8("requête : ") << requete;

    retour = bdd->recuperer(requete, seuils);
    if(retour != false)
    {
        qDebug() << QString::fromUtf8("seuil : entre %1 et %2").arg(seuils.at(0)).arg(seuils.at(1));
```

 © 2019 tv <tvaira@free.fr>

```
    }
    else
    {
        qDebug() << QString::fromUtf8("erreur !");
    }

    // Exemple : récupère au moins deux champs de plusieurs enregistrements
    // les enregistrements sont récupérés dans un QVector de QStringList

    // récupère les dépassement de seuil
    requete = "SELECT date, heure, temperature FROM mesures WHERE temperature >= '" + seuils.at(1) + "' OR
    temperature <= '" + seuils.at(0) + "' ORDER BY heure ASC";
    qDebug() << QString::fromUtf8("requête : ") << requete;

    retour = bdd->recuperer(requete, listeAlarmes);
    if(retour != false)
    {
        for(int i=0; i < listeAlarmes.size(); i++)
        {
            alarme = listeAlarmes.at(i);
            qDebug() << QString::fromUtf8("%1 %2 : %3").arg(alarme.at(0)).arg(alarme.at(1)).arg(alarme.at(2));
        }
    }
    else
    {
        qDebug() << QString::fromUtf8("erreur !");
    }

    // Exemple : INSERTION d'un enregistrement
    QString dateDebut = "2015-06-17";
    QString heureDebut = "14:03:00";
    float temperature = 36.5;

    requete = "INSERT INTO mesures (id, date, heure, temperature) VALUES ('', '" + dateDebut + "', '" +
    heureDebut + "', '" + QString::number(temperature) + "')";
    qDebug() << QString::fromUtf8("requête : ") << requete;
    retour = bdd->executer(requete);
    if(retour == false)
    {
        qDebug() << QString::fromUtf8("erreur insertion !");
    }

    // Exemple : MODIFICATION d'un enregistrement
    int nouvellePeriode = periode.toInt() * 10;
    requete = "UPDATE parametres SET periode='" + QString::number(nouvellePeriode) + "'";
    qDebug() << QString::fromUtf8("requête : ") << requete;
    retour = bdd->executer(requete);
    if(retour == false)
    {
        qDebug() << QString::fromUtf8("erreur modification !");
    }

    // Exemple : SUPPRESSION d'un enregistrement
    requete = "DELETE FROM mesures WHERE id='1'";
    qDebug() << QString::fromUtf8("requête : ") << requete;
    retour = bdd->executer(requete);
    if(retour == false)
    {
        qDebug() << QString::fromUtf8("erreur suppression !");
    }

    BaseDeDonnees::detruireInstance();

    return 0;
}
```

 © 2019 tv <tvaira@free.fr>

Code source : test-mo-bd.zip

Voir aussi :

- Schéma relationnel
- Base de données sous Android [PDF]

Retour au sommaire

 © 2019 tv <tvaira@free.fr>