9. Accès à une base de données avec Qt

Qt offre un module dédié pour interagir avec les bases de données relationnelles : **Qt SQL Module**. Les classes principales sont :

- **QSqlDatabase** : Permet de se connecter à une base de données.
- **QSqlQuery** : Gère l'exécution de requêtes SQL.
- **QSqlError** : Fournit des informations sur les erreurs liées à la base de données.
- **QSqlTableModel** et **QSqlQueryModel** : Modèles utilisés pour afficher des données dans des vues Qt, comme QTableView.

9.1. Configuration de l'environnement

- **Ajout du module Qt SQL**

  Dans le fichier .pro du projet il est nécessaire d'ajouter : `QT += sql`

- **Vérification de la prise en charge des pilote pour interagir avec les bases de données**

  Qt utilise des pilotes spécifiques pour interagir avec les différents systèmes de gestion de bases de données. Le code suivant permet de vérifier les pilotes disponibles dans l'environnement de développement :

Vérification des pilotes pris en charge

```
#include <QSqlDatabase>
qDebug() << QsqlDatabase::drivers();
// exemple de résultat : QList("QODBC", "QPSQL", "QMARIADB", "QMYSQL", "QSQLITE")
```

Remarque : Si le pilote attendu n'apparaît pas, il doit être installé ou compilé en fonction de l'environnement.

Pour les exemples qui suivent, la classe `AccesBdd` possède un attribut privé : `QSqlDatabase db;`

- **Connexion et déconnexion avec la base de données**

  Pour se connecter à une base de données, il faut disposer de l'adresse du serveur, du nom de la base et du login et du mot de passe.

Connexion à la base de données

```
#include <QSqlDatabase>
#include <QSqlError>

bool AccesBdd::SeConnecterAuSGBdd() {
    bool retour = false;
    db = QSqlDatabase::addDatabase("QMARIADB"); // ou "QMYSQL"
    db.setHostName("localhost");    // Adresse du serveur MySQL
    db.setDatabaseName("nom_base"); // Nom de la base de données
    db.setUserName("utilisateur");  // Nom d'utilisateur
    db.setPassword("mot_de_passe"); // Mot de passe
    if(db.open()){
        qDebug() << "Connexion réussie à la base de données.";
        retour = true;
    }
    else
        qDebug() << "Erreur lors de la connexion :" << db.lastError().text();

    return retour;
}
```

Une bonne pratique pour la déconnexion consiste à fermer la connexion et à libérer les ressources associées. L'objectif de la fermeture empêchent toutes requêtes ou opérations. La libération des ressources restitue la mémoire allouée pour gérer cette connexion.

Déconnexion de la base de données

```
void AccesBdd::SeDeconnecterDuSGBdd() {

    db.close();
    Qtimer::singleShot(100,this, [=]() {  // Délai de 100ms avant de retirer la base
        QSqlDatabase::removeDatabase(QSqlDatabase::defaultConnection);
    });
}
```

9.2. Création et exécution de requêtes

Après avoir établie une connexion avec la base de données en utilisant `QSqlDatabase` les transactions sont réalisées par des requêtes SQL.

`QSqlQuery` est donc une classe essentielle de Qt pour exécuter ces requêtes. Elle permet d'exécuter des commandes, comme `SELECT`, `INSERT`, `UPDATE`, `DELETE`, etc. Deux types de requêtes peuvent être utilisées.

9.2.1. Exécution de requêtes SQL simples

Les requêtes simples peuvent être exécutées avec la méthode `exec()`. Le paramètre fourni à la fonction constitue la requête SQL.

Exemple de requête simple

```
#include <QSqlQuery>
#include <QSqlError>

void AccesBdd::CreerUneTable() {
    if(db.isOpen())
    {
        QSqlQuery query("CREATE TABLE IF NOT EXISTS exemple (id INT PRIMARY KEY, nom VARCHAR(50))");
        if (!query.exec()) {
            qDebug() << "Erreur lors de la création de la table :" << query.lastError().text();
        } else
            qDebug() << "Table créée ou déjà existante.";
    }
    else qDebug() << "La base de données n'est pas ouverte";
}
```

Il arrive parfois que la requête soit passée directement à la méthode exec(), cette technique est cependant considérée obsolète depuis la version 6.6 de Qt, elle pourrait être amenée à disparaître dans de futures versions.

9.2.2. Utilisation des requêtes préparées

Les requêtes préparées sont recommandées pour accroître la sécurité, Elles préviennent des injections SQL. Elles permettent de définir des paramètres dynamiques via des `bindValue()` ce qui permet de gagner également en performance. Une requête préparée est compilée une seule fois et peut être réutilisée

Exemple de requête simple

```
#include <QSqlQuery>
#include <QSqlError>

void AccesBdd::Consulter(const int _id)
{
    QSqlQuery query;
    if(_id != 0) {
        query.prepare("SELECT * FROM personnes WHERE id = :id;");
        query.bindValue(":id",_id);
    }
    else query.prepare("SELECT * FROM personnes");

    if(query.exec()) {
        qDebug() << "Nombre d'éléments trouvés : " << query.size();
        while(query.next()) {
            int id = query.value("id").toInt();
            QString nom = query.value("nom").toString();
            int age = query.value("age").toInt();

            qDebug() << "Id : " << id << " nom : " << nom << " age : " << age;
        }
    }
    else qDebug() << "Erreur lors l'opération de sélection : " << query.lastError().text();
}
```

9.2.3. Préparation de la requête

Une requête préparée utilise des paramètres dynamiques pour insérer ou filtrer des données. Ces paramètres peuvent être :

- **Nommés** (plus lisibles) :

```
query.prepare("SELECT * FROM personnes WHERE id = :id");
query.bindValue(":id", _id);
```

- **Positionnels** (moins explicites mais fonctionnels) :

```
query.prepare("SELECT * FROM personnes WHERE id = ?");
query.addBindValue(_id);
```

Dans ce cas le caratère '?' Indique la position du paramètres les paramètres sont ensuites ajoutés dans l'ordre d'apparition.

9.2.4. Obtention des résultats

Une fois une requête exécutée avec succès, les résultats peuvent être parcourus ligne par ligne grâce à la méthode **`next()`**, et chaque colonne peut être récupérée via **`value()`**. Pour cette dernière méthode la récupération de la valeur peut-être faite :

- Par nom de colonne : `query.value("nom")`).
- Par index de colonne : `query.value(1)`).

La méthode `size()` retourne le nombre de résultats de la requête, mais elle peut ne pas être prise en charge par certains systèmes de bases de données. Si elle n'est pas supportée, elle retourne `-1`.

9.2.5. Optimisations et astuces

- **Préparer une seule fois les requêtes répétées :** Pour des requêtes exécutées fréquemment avec des paramètres différents, il est plus performant de préparer la requête une seule fois. La méthode `bindValue()` est utilisée pour actualiser les paramètres avant chaque exécution. Les résultats précédents sont écrasés.

Exemple de transaction

```
void AccesBdd::AjouterPlusieursUtilisateurs(const QVector<QPair<QString, int>> & users)
{
    QSqlQuery query;
    query.prepare("INSERT INTO personnes (nom, age) VALUES (:nom, :age)");

    for (const auto& utilisateur : users) {
        query.bindValue(":nom", utilisateur.first);
        query.bindValue(":age", utilisateur.second);

        if (!query.exec()) {
            qDebug() << "Erreur ajout de l'utilisateur :" << query.lastError().text();
        }
    }
}
```

- **Utiliser des transactions pour les opérations complexes :** Une transaction permet de regrouper plusieurs opérations afin qu'elles soient exécutées ou annulées ensemble en cas d'erreur.

Exemple de transaction

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

QSqlDatabase db = QSqlDatabase::database();
db.transaction();
QSqlQuery query;
query.exec("INSERT INTO personnes (nom, age) VALUES ('Alice', 25)");
if (!db.commit()) {
    qDebug() << "Échec de la transaction :" << db.lastError().text();
}
```

- **Gestion des erreurs avec précision :** Pour déboguer efficacement les erreurs SQL :
  - **`query.lastError()`** : Retourne les détails de l'erreur.
  - **`query.lastQuery()`** : Retourne la dernière requête exécutée.
- **Libération des ressources :** Il est conseillé de libérer les ressources associées aux objets `QSqlQuery` après chaque utilisation en appelant la méthode **`finish()`**.
- **Test de l'ouverture de la base** : Il est bien également de vérifier que la l'accès à base de données est ouverte avant d'exécuter une requête afin de limiter les erreurs.

Test d'ouverture

```
if (!db.isOpen()) {
    qDebug() << "Base non ouverte.";
}
else {
 // fabrication et exécution de la requêtes
}
```

9.3. Traitement des erreurs en utilisant les exceptions

9.3.1. Qt et le traitement d'erreurs de la base de données

Les exceptions offrent un moyen structuré et centralisé de gérer les erreurs. Contrairement à d'autres mécanismes, comme les codes de retour ou des valeurs spéciales, elles permettent de séparer clairement le flux normal d'exécution du traitement des erreurs. Cela évite la répétition de blocs similaires dans plusieurs fonctions. Le module SQL de Qt ne lève pas d'exceptions afin de privilégier une compatibilité large avec des langages et plateformes où les exceptions ne sont pas toujours supportées. Les développeurs peuvent choisir d'utiliser des exceptions ou d'autres mécanismes, comme les codes d'erreur, selon leurs besoins à partir des méthodes comme :

- `QSqlQuery::exec()` qui retourne un booléen pour indiquer un succès ou un échec.
- `QSqlQuery::lastError()` qui fournit des informations détaillées sur l'erreur.

9.3.2. Création d'un gestionnaire d'exceptions personnalisées

Pour intégrer un mécanisme basé sur les exceptions avec le module Qt SQL, Une classe dérivée de `QException` peut être crée :

Classe de gestion pour les execptions

```
#ifndef DATABASEEXCEPTION_H
#define DATABASEEXCEPTION_H

#include <QException>
#include <QString>

class DatabaseException : public QException {
public:

    explicit DatabaseException(const QString& message)
        : errorMessage(message) {}
    void raise() const override { throw *this; }
    QException* clone() const override { return new DatabaseException(*this); }
    QString getMessage() const { return errorMessage; }

private:
    QString errorMessage;

};
#endif // DATABASEEXCEPTION_H
```

Ce mécanisme de gestionnaire d'exception peut-être facilement addapté à d'autre type d'application.

Les différentes méthodes pour la gestion de la base de données peuvent être modifiée pour utilser une gestion d'erreur utilisant les exceptions. Par exemple pour l'ajout d'un utilisateur le code correspondant peut-être :

Classe de gestion pour les execptions

```
void AccesBdd::AjouterUtilisateur(const Qstring & _nom,const int _age) {
    QSqlQuery query;
    query.prepare("INSERT INTO users (nom, age) VALUES (:nom, :age)");
    query.bindValue(":nom", _nom);
    query.bindValue(":age", _age);

    if (!query.exec()) {
        throw DatabaseException("Erreur lors de l'ajout d'un utilisateur : " +
                                query.lastError().text());
    }
}
```

Le programme appelant est dans le cas d'un traitement d'erreur utilisant les exceptions :

Côté interface utilisateur

```
void MainWindow::on_pushButton_AjouterUtilisateur_Clicked() {
    QString nom = ui->lineEditNom->text();
    int age = ui->spinBoxAge->value();

    try {
        accesBdd->AjouterUtilisateur(nom, age);
        QMessageBox::information(this, "Succès", "Utilisateur ajouté avec succès.");
    } catch (const DatabaseException& ex) {
        QMessageBox::critical(this, "Erreur", ex.getMessage());
    }
}
```

9.4. Affichage des résultats dans une interface

Lorsque l'affichage des résultats se fait sous forme de tableau, L'interface utilisateur peut posséder une **QTableView**, la classe **QSqlQueryModel** dans ce cas simplifie le traitement des données lors de l'utilisation de l'architecture **Model-View-Controller**.

Dans le gestionnaire de base de données

```
QSqlQueryModel* AccesBdd::ObtenirPersonnes()
{
    QSqlQueryModel* model = new QSqlQueryModel();
    model->setQuery("SELECT id, nom, age FROM personnes");
    if (model->lastError().isValid())
       qDebug() << "Erreur récupération des données :" << model->lastError().text();
    return model;
}
```

Pour la partie utilisateur le traitement est également relativement simple

Côté interface utilisateur

```
void MainWindow::AffichageDesPersonnes()
{
    QSqlQueryModel* model = dbManager->getPersonnes();
    ui->tableView->setModel(model);
}
```

9.5. Gestion Simplifiée des Tables SQL avec QsqlTableModel

La classe **`QSqlTableModel`** de Qt est un modèle de données conçu pour afficher et manipuler les données d'une table dans une base de données relationnelle. Elle s'intègre directement avec les widgets de type **`QTableView`**, permettant une interaction fluide entre l'interface utilisateur et la base de données.

9.5.1. Affichage automatique des données

La classe QSqlTableModel permet d'extraire les données directement depuis une table SQL et les affiche dans un widget **QTableView** ou similaire. elle est conçue pour fonctionner avec une table entière ou un sous-ensemble filtré des données.

Les données modifiées dans l'interface utilisateur en utilisant une `QTableView` sont automatiquement mises à jour dans la base de données, sauf si cette fonctionnalité est désactivée. Les suppressions et ajouts sont également pris en charge.

Utilisation de QSqlTableModel

```
QSqlTableModel* model = new QSqlTableModel(this);
model->setTable("personnes");
model->select();
ui->tableView->setModel(model);
```

La méthode setTable() définit la table à utiliser, select() charge les données de la table dans le modèle.

9.5.2. Tri et filtrage

Le tri et le filtrage des données peuvent être effectués facilement avec des méthodes comme setFilter et setSort.

Utilisation de QSqlTableModel

```
model->setFilter("age > 30");
model->select();

model->setSort(1, Qt::AscendingOrder); // 1 = index de la colonne "nom"
model->select();
```

9.5.3. Édition des données

La méthode `setEditStrategy()` permet de définir comment les modifications apportées aux données sont traitées en fonction de la constante :

- **QSqlTableModel::OnFieldChange**

  Les modifications sont appliquées immédiatement à la base de données après chaque modification d'une cellule. Aucun bouton supplémentaire n'est nécessaire.
- **QSqlTableModel::OnRowChange**

  Les modifications sont appliquées à la base de données uniquement lorsque l'utilisateur quitte la ligne modifiée. Aucun bouton n'est nécessaire.
- **QSqlTableModel::OnManualSubmit**

  Les modifications sont conservées localement jusqu'à ce que l'utilisateur valide explicitement (via submitAll()). Un bouton supplémentaire est nécessaire pour valider les modifications, et un autre peut être ajouté pour les annuler (via revertAll()).

Validation manuel des modification

```
void MainWindow::on_PushButton_Submit_Clicked() {
    if (model->submitAll())
        QMessageBox::information(this, "Succès", "Modifications enregistrées.");
    else
        QMessageBox::critical(this, "Erreur", "Échec de l'enregistrement : "
                              + model->lastError().text());
}

void MainWindow::on_PushButton_Annuler_Clicked() {
    model->revertAll();
    QMessageBox::information(this, "Info", "Modifications annulées.");
}
```

9.5.4. Exemple d'utilisation

Implémentation de la classe de Visualisation de la BDD

```
#include "widget.h"
#include "ui_widget.h"
#include <QSqlDatabase>
#include <QSqlQuery>
#include <QSqlError>
#include <QMessageBox>
Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
    ,model(nullptr)
{
    ui->setupUi(this);
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
    db.setHostName("172.18.58.7");
    db.setDatabaseName("CoursBdd");
    db.setUserName("snir");
    db.setPassword("snir");
    if (db.open()){
        model = new QSqlTableModel(this);
        model->setTable("personnes");
        model->setEditStrategy(QSqlTableModel::OnManualSubmit); // Validation manuelle
        model->select();
        // Lier le modèle à la vue
        ui->tableView->setModel(model);
        ui->tableView->horizontalHeader()->setSectionResizeMode(QHeaderView::Stretch);
    }
    else QMessageBox::critical(this, "Erreur", "Connexion à la base impossible.");
}
Widget::~Widget()
{
    delete ui;
}
void Widget::on_pushButtonValider_clicked()
{
    if (model->submitAll())
        QMessageBox::information(this, "Succès", "Modifications enregistrées.");
    else
        QMessageBox::critical(this, "Erreur", "Échec enregistrement : "
                              + model->lastError().text());
}



void Widget::on_pushButtonAnuler_clicked()
{
    model->revertAll();
    QMessageBox::information(this, "Info", "Modifications annulées.");
}
void Widget::on_pushButtonAjouter_clicked()
{
    int row = model->rowCount();
    model->insertRow(row);
    QMessageBox::information(this, "Info", "Nouvelle ligne ajoutée.");
}
void Widget::on_pushButtonSupprimer_clicked()
{
    QModelIndexList selection = ui->tableView->selectionModel()->selectedRows();
    for (const QModelIndex& index : selection)
        model->removeRow(index.row());
    model->submitAll(); // Valider les suppressions
    QMessageBox::information(this, "Info", "Ligne(s) supprimée(s).");
}
```

Définition de la classe de Visualisation de la BDD

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>
#include <QSqlTableModel>
QT_BEGIN_NAMESPACE
namespace Ui {
class Widget;
}
QT_END_NAMESPACE
class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = nullptr);
    ~Widget();
private slots:
    void on_pushButtonValider_clicked();
    void on_pushButtonAnuler_clicked();
    void on_pushButtonAjouter_clicked();
    void on_pushButtonSupprimer_clicked();
private:
    Ui::Widget *ui;
    QSqlTableModel* model;
};
#endif // WIDGET_H
```