Creating database driven applications using Qt

This article demonstrates how you can integrate SQLite features to your application using the model/view architecture in a "contacts" demo application.

25 Jul 2010

Introduction

Another great feature in Qt is its support for Structured Query Language (SQL). Using the Qt model/view architecture and minimum SQL knowledge it is possible to create powerful database driven applications. If you are a SQL beginner you can find several good introductory material about SQL just searching in the web. SQL is considered the most used language for relational database management systems (RDBMS) and it is part of any computer science course. In special, the SQLite back-end is useful for mobile application since it is a self-contained, server-less, zero-configuration, transactional SQL database engine. SQLite has been used in many large projects like Mozilla Firefox, Symbian smartphones, Skype, Solaris, PHP, Mac OS, among others, and it is probably the most widely deployed SQL database. SQLite back-end in Qt is available for desktop, Meego and Symbian.

In this article will be demonstrated how you can integrate SQLite features to your application using the model/view architecture in a "contacts" demo application. The Qt SDK was used in this example.

If you are using Maemo devices like N900, it is necessary to install SQLite for Qt before testing this application in your device. Open the **X Terminal** application and type following commands:

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

sudo gainroot enables you to execute commands in your phone as administrator and it needs to be used with care.

For Symbian users, the application **sqlite3.sis** is required. It can be found inside your Nokia Qt SDK at **\NokiaQtSDK\Symbian\sis\sqlite3.sis** or ready for installation from Start menu (Nokia Qt SDK -> Symbian -> Install SQLite to Symbian).

If you do not want to use a real device, an alternative is to use the Qt Simulator target, available in the Nokia Qt SDK. No additional installations are required.

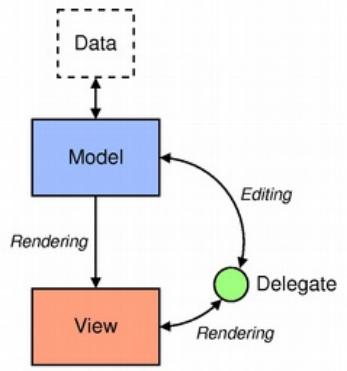
Qt Classes for accessing SQLite

Using Qt, SQL classes are divided into three layers:

- **Driver layer**: a set of useful classes for those that want to create your own database driver or add some customization to existing ones.
- **SQL API layer**: basic access to database, like connections, errors, field manipulation, and so on.
- **User Interface layer**: a set of classes to be used with Qt's model/view architecture.

Only second and third layers will be used in this article, represented by the following classes available in the model/view architecture:

- **QSqlDatabase**: the database connector (the data source, for the model/view). Since SQLite is mapped into an archive, the connection is straightforward. For more sophisticated databases, parameters like login, password and ports can be provided.
- **QSqlTableModel**: used as model, it will provide a high-level interface for accessing the database.
- **QTreeView**: the view object used to show database records.



[1]

Connecting the SQLite database to Qt's model/view architecture

As SQLite databases are mapped on files, the connection to the database consists only on specifying the file to be used. Of course, the path to the file can be different if you are programming for MeeGo or for Symbian and you should foreseen this situation with #ifdef Q_OS_SYMBIAN statements. Moreover, you

(C) Copyright Nokia 2013. All rights reserved.

will need to create the database structure inside this file before using it for storing registers.

In the following code snippet the database connector is created and the database opened. If the database structure does not exist yet it will be created.

```
// using SQLITE driver
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName("path_to_database_file");

if (db.open()) {
    // Try to locate the contacts database.
    // If it is not available create it.
    if (db.tables().indexOf("contacts") == -1) {
        QSqlQuery query(db);
        query.prepare("CREATE TABLE contacts (name VARCHAR(32), phone VARCHAR(16))");
        if (!query.exec())
            qCritical() << query.lastError();
    }
} else {
    qCritical() << db.lastError();
}
```

The next step is to associate this database to the model/view architecture. **setTable()** (from model class) and **setModel()** (from view class) methods are used to accomplish this task. It is necessary to call **select()** to populate the model with data from table and programmer can choose his edit strategy calling **setEditEstrategy()**. In this case, using **QSqlTableModel::OnFieldChange**, any change will be applied immediately to the database.

```
// create a new model and use the contacts table
QSqlTableModel *model = new QSqlTableModel(this, db);
model->setTable("contacts");
// All changes to the model will be applied immediately to the database
model->setEditStrategy(QSqlTableModel::OnFieldChange);
// Populate the model with data from table
model->select();
// creating the model/view association
ui->view->setModel(model);
```

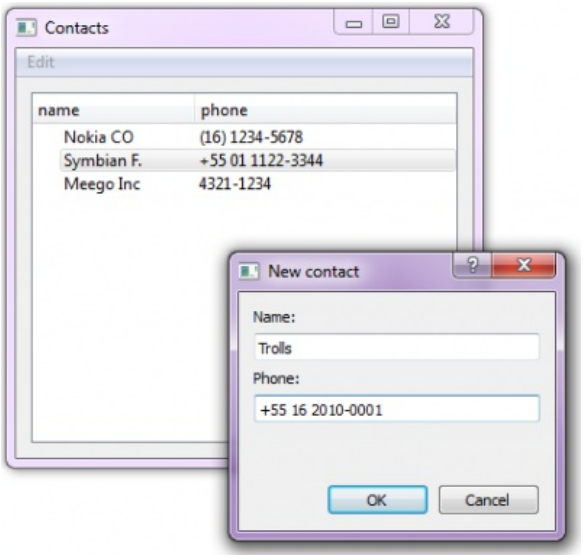
Now your application is ready to startup and all database data will be available to the user though the view component. Additionally, your application has support for data persistence without any line of code, all registers are kept between successive runs of your program.

Adding registers

Once registers can be viewed, your next task is to create a dialog for gathering data from users. Form designer in Qt Creator makes this task very easy. A simple dialog with only two fields (name and telephone) are necessary for this demo application. Create this dialog and the related classes and add them to your project, adding callbacks for **Cancel**and **Ok** buttons. To show up the dialog, you need to use the `exec()` method. After dialog is closed, you can check if you have valid data and thus add it to your database as a new register. The **record()** (from model) method will return a SQL record that matches to your database specification and you can fill this register using `setValue()` method. Qt properties were used in the dialog class to help us to collect the necessary data and the model method `insertRecord()` will add this register to the database.

```
// Create a new empty dialog and show it using exec()
NewContact dlg(this);
dlg.resetName();
dlg.resetPhone();

if(dlg.exec() == QDialog::Accepted && dlg.name().length() > 0) {
    // create a record from the current model
    QSqlRecord rec = model->record();
    rec.setValue("name", dlg.name());
    rec.setValue("phone", dlg.phone());
    // insert a new record (-1)
    model->insertRecord(-1,rec);
}
```



Deleting registers

(C) Copyright Nokia 2013. All rights reserved.

Register deletion is simple since you have the register's index. Get the index as a `QModelIndex` object querying your view component and use it as parameter for a call to `removeRow()` (from model).

```
// get the current index, if any
QModelIndex sample = ui->view->currentIndex();
if (sample.row() >= 0) {
    QMessageBox::StandardButton reply;
    reply = QMessageBox::question(this, tr("Remove contact"),
                                  QString(tr("Remove contact ?")),
                                  QMessageBox::Yes | QMessageBox::No);

    if(reply == QMessageBox::Yes)
        model->removeRow(sample.row()); // remove the current index
}
```

Modifying registers

After creating `addRegister()` and `deleteRecord()` methods, it is easier to code the `editRecord()` method. Get the register's index and fill the dialog using information from it. Thus present this dialog to the user and save changes.

```
// get the current index, if any
QModelIndex sample = ui->view->currentIndex();
if (sample.row() >= 0 ) {
    // copy the current record
    QSqlRecord rec = model->record(sample.row());
    NewContact dlg(this);
    dlg.setName(rec.value("name").toString());
    dlg.setPhone(rec.value("phone").toString());

    if (dlg.exec() == QDialog::Accepted) {
        rec.setValue("name",dlg.name());
        rec.setValue("phone",dlg.phone());
        // save modified data
        model->setRecord(sample.row(),rec);
    }
}
```

Conclusion

With Qt it is possible to create concise programs that use SQL data persistence using few lines of codes. In special, SQLite abstraction provided by Qt is really straightforward when used together Qt's model/view architecture.

Source code

This demo program was tested in Maemo, Linux and Windows operating systems without modifications. You can download the entire source code from this link.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.