C++ GUI Programming with Qt4: Databases

By Jasmin Blanchette and Mark Summerfield

Nov 6, 2009

Sample Chapter is provided courtesy of Prentice Hall

This chapter covers accessing SQL databases for connecting and querying, viewing tables, editing records using forms, and presenting data in tabular forms.

This chapter is from the book
C++ GUI Programming with Qt4, 2nd Edition

13. Databases

- *Connecting and Querying*
- *Viewing Tables*
- *Editing Records Using Forms*
- *Presenting Data in Tabular Forms*

The *QtSql* module provides a platform- and database-independent interface for accessing SQL databases. This interface is supported by a set of classes that use Qt's model/view architecture to provide database integration with the user interface. This chapter assumes familiarity with Qt's model/view classes, covered in Chapter 10.

A database connection is represented by a `QSqlDatabase` object. Qt uses drivers to communicate with the various database APIs. The Qt Desktop Edition includes the following drivers:

Driver	Database
QDB2	*IBM DB2 version 7.1and later*
QIBASE	*Borland InterBase*
QMYSQL	*MySQL*
QOCI	*Oracle (Oracle Call Interface)*
QODBC	*ODBC (includes Microsoft SQL Server)*
QPSQL	*PostgreSQL 7.3 and later*
QSQLITE	*SQLite version 3*
QSQLITE2	*SQLite version 2*
QTDS	*Sybase Adaptive Server*

Due to license restrictions, not all of the drivers are provided with the Qt Open Source Edition. When configuring Qt, we can choose between including the SQL drivers inside Qt itself and building them as plugins. Qt is supplied with the SQLite database, a public domain in-process database. [*]

For users who are comfortable with SQL syntax, the `QSqlQuery` class provides a means of directly executing arbitrary SQL statements and handling their results. For users who prefer a higher-level database interface that avoids SQL syntax, `QSqlTableModel` and `QSqlRelationalTableModel` provide suitable abstractions. These classes represent an SQL table in the same way as Qt's other model classes (covered in Chapter 10). They can be used stand-alone to traverse and edit data in code, or they can be attached to views through which end-users can view and edit the data themselves.

Qt also makes it straightforward to program the common database idioms, such as master–detail and drill-down, and to view database tables using forms or GUI tables, as the examples in this chapter will demonstrate.

Connecting and Querying

To execute SQL queries, we must first establish a connection with a database. Typically, database connections are set up in a separate function that we call at application startup. For example:

```
bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
    db.setHostName("mozart.konkordia.edu");
    db.setDatabaseName("musicdb");
    db.setUserName("gbatstone");
    db.setPassword("T17aV44");
    if (!db.open()) {
        QMessageBox::critical(0, QObject::tr("Database Error"),
                              db.lastError().text());
        return false;
    }
    return true;
}
```

First, we call `QSqlDatabase::addDatabase()` to create a `QSqlDatabase` object. The first argument to `addDatabase()` specifies which database driver Qt must use to access the database. In this case, we use MySQL.

Next, we set the database host name, the database name, the user name, and the password, and we open the connection. If `open()` fails, we show an error message.

Typically, we would call `createConnection()` in `main()`:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (!createConnection())
        return 1;
    ...
    return app.exec();
}
```

Once a connection is established, we can use `QSqlQuery` to execute any SQL statement that the underlying database supports. For example, here's how to execute a `SELECT` statement:

```
QSqlQuery query;
query.exec("SELECT title, year FROM cd WHERE year >= 1998");
```

After the `exec()` call, we can navigate through the query's result set:

```
while (query.next()) {
    QString title = query.value(0).toString();
    int year = query.value(1).toInt();
    std::cerr << qPrintable(title) << ": " << year << std::endl;
}
```

We call `next()` once to position the `QSqlQuery` on the *first* record of the result set. Subsequent calls to `next()` advance the record pointer by one record each time, until the end is reached, at which point `next()` returns `false`. If the result set is empty (or if the query failed), the first call to `next()` will return `false`.

The `value()` function returns the value of a field as a `QVariant`. The fields are numbered from 0 in the order given in the `SELECT` statement. The `QVariant` class can hold many C++ and Qt types, including `int` and `QString`. The different types of data that can be stored in a database are mapped into the corresponding C++ and Qt types and stored in `QVariant`s. For example, a `VARCHAR` is represented as a `QString` and a `DATETIME` as a `QDateTime`.

`QSqlQuery` provides some other functions to navigate through the result set: `first()`, `last()`, `previous()`, and `seek()`. These functions are convenient, but for some databases they can be slower and more memory-hungry than `next()`. For an easy optimization when operating on large data sets, we can call `QSqlQuery::setForwardOnly(true)` before calling `exec()`, and then only use `next()` for navigating through the result set.

Earlier we specified the SQL query as an argument to `QSqlQuery::exec()`, but we can also pass it directly to the constructor, which executes it immediately:

```
QSqlQuery query("SELECT title, year FROM cd WHERE year >= 1998");
```

We can check for an error by calling `isActive()` on the query:

```
if (!query.isActive())
    QMessageBox::warning(this, tr("Database Error"),
                         query.lastError().text());
```

If no error occurs, the query will become "active" and we can use `next()` to navigate through the result set.

Doing an `INSERT` is almost as easy as performing a `SELECT`:

```
QSqlQuery query("INSERT INTO cd (id, artistid, title, year) "
                "VALUES (203, 102, 'Living in America', 2002)");
```

After this, `numRowsAffected()` returns the number of rows that were affected by the SQL statement (or -1 on error).

If we need to insert a lot of records, or if we want to avoid converting values to strings (and escaping them correctly), we can use `prepare()` to specify a query that contains placeholders and then bind the values we want to insert. Qt supports both the Oracle-style and the ODBC-style syntax for placeholders for all databases, using native support where it is available and simulating it otherwise. Here's an example that uses the Oracle-style syntax with named placeholders:

```
QSqlQuery query;
query.prepare("INSERT INTO cd (id, artistid, title, year) "
              "VALUES (:id, :artistid, :title, :year)");
query.bindValue(":id", 203);
query.bindValue(":artistid", 102);
query.bindValue(":title", "Living in America");
query.bindValue(":year", 2002);
query.exec();
```

Here's the same example using ODBC-style positional placeholders:

```
QSqlQuery query;
query.prepare("INSERT INTO cd (id, artistid, title, year) "
              "VALUES (?, ?, ?, ?)");
query.addBindValue(203);
query.addBindValue(102);
query.addBindValue("Living in America");
query.addBindValue(2002);
query.exec();
```

After the call to `exec()`, we can call `bindValue()` or `addBindValue()` to bind new values, and then call `exec()` again to execute the query with the new values.

Placeholders are often used to specify binary data or strings that contain non-ASCII or non-Latin-1 characters. Behind the scenes, Qt uses Unicode with those databases that support Unicode, and for those that don't, Qt transparently converts strings to the appropriate encoding.

Qt supports SQL transactions on databases where they are available. To start a transaction, we call `transaction()` on the `QSqlDatabase` object that represents the database connection. To finish the transaction, we call either `commit()` or `rollback()`. For example, here's how we would look up a foreign key and execute an `INSERT` statement inside a transaction:

```
QSqlDatabase::database().transaction();
QSqlQuery query;
query.exec("SELECT id FROM artist WHERE name = 'Gluecifer'");
if (query.next()) {
    int artistId = query.value(0).toInt();
    query.exec("INSERT INTO cd (id, artistid, title, year) "
               "VALUES (201, " + QString::number(artistId)
               + ", 'Riding the Tiger', 1997)");
}
QSqlDatabase::database().commit();
```

The `QSqlDatabase::database()` function returns a `QSqlDatabase` object representing the connection we created in `createConnection()`. If a transaction cannot be started, `QSqlDatabase::transaction()` returns `false`. Some databases don't support transactions. For those, the `transaction()`, `commit()`, and `rollback()` functions do nothing. We can test whether a database supports transactions using `hasFeature()` on the `QSqlDriver` associated with the database:

```
QSqlDriver *driver = QSqlDatabase::database().driver();
if (driver->hasFeature(QSqlDriver::Transactions))
    ...
```

Several other database features can be tested for, including whether the database supports BLOBs (binary large objects), Unicode, and prepared queries.

It is also possible to access the low-level database driver handle and the low-level handle to a query's result set, using `QSqlDriver::handle()` and `QSqlResult::handle()`. However, both functions are dangerous unless you know exactly what you are doing and are very careful. See their documentation for examples and an explanation of the risks.

In the examples so far, we have assumed that the application is using a single database connection. If we want to create multiple connections, we can pass a name as a second argument to `addDatabase()`. For example:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL", "OTHER");
db.setHostName("saturn.mcmanamy.edu");
db.setDatabaseName("starsdb");
db.setUserName("hilbert");
db.setPassword("ixtapa7");
```

We can then retrieve a pointer to the `QSqlDatabase` object by passing the name to `QSqlDatabase::database()`:

```
QSqlDatabase db = QSqlDatabase::database("OTHER");
```

To execute queries using the other connection, we pass the `QSqlDatabase` object to the `QSqlQuery` constructor:

```
QSqlQuery query(db);
query.exec("SELECT id FROM artist WHERE name = 'Mando Diao'");
```

Multiple connections are useful if we want to perform more than one transaction at a time, since each connection can handle only a single active transaction. When we use multiple database connections, we can still have one unnamed connection, and `QSqlQuery` will use that connection if none is specified.

In addition to `QSqlQuery`, Qt provides the `QSqlTableModel` class as a higher-level interface, allowing us to avoid using raw SQL for performing the most common SQL operations (`SELECT`, `INSERT`, `UPDATE`, and `DELETE`). The class can also be used stand-alone to manipulate a database without any GUI involvement, or it can be used as a data source for `QListView` or `QTableView`.

Here's an example that uses `QSqlTableModel` to perform a `SELECT`:

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("year >= 1998");
model.select();
```

This is equivalent to the query

```
SELECT * FROM cd WHERE year >= 1998
```

Navigating through the result set is done by retrieving a given record using `QSqlTableModel::record()` and by accessing individual fields using `value()`:

```
for (int i = 0; i < model.rowCount(); ++i) {
    QSqlRecord record = model.record(i);
    QString title = record.value("title").toString();
    int year = record.value("year").toInt();
    std::cerr << qPrintable(title) << ": " << year << std::endl;
}
```

The `QSqlRecord::value()` function takes either a field name or a field index. When operating on large data sets, it is recommended that fields are specified by their indexes. For example:

```
int titleIndex = model.record().indexOf("title");
int yearIndex = model.record().indexOf("year");
for (int i = 0; i < model.rowCount(); ++i) {
    QSqlRecord record = model.record(i);
    QString title = record.value(titleIndex).toString();
    int year = record.value(yearIndex).toInt();
    std::cerr << qPrintable(title) << ": " << year << std::endl;
}
```

To insert a record into a database table, we call `insertRow()` to create a new empty row (record), and we use `setData()` to set the values of each column (field):

```
QSqlTableModel model;
model.setTable("cd");
int row = 0;
model.insertRows(row, 1);
model.setData(model.index(row, 0), 113);
model.setData(model.index(row, 1), "Shanghai My Heart");
model.setData(model.index(row, 2), 224);
model.setData(model.index(row, 3), 2003);
model.submitAll();
```

After the call to `submitAll()`, the record might be moved to a different row position, depending on how the table is ordered. The `submitAll()` call will return `false` if the insertion failed.

An important difference between an SQL model and a standard model is that for an SQL model we must call `submitAll()` to have any changes written to the database.

To update a record, we must first position the `QSqlTableModel` on the record we want to modify (e.g., using `select()`). We then extract the record, update the fields we want to change, and write our changes back to the database:

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("id = 125");
model.select();
if (model.rowCount() == 1) {
    QSqlRecord record = model.record(0);
    record.setValue("title", "Melody A.M.");
    record.setValue("year", record.value("year").toInt() + 1);
    model.setRecord(0, record);
    model.submitAll();
}
```

If there is a record that matches the specified filter, we retrieve it using `QSqlTableModel::record()`. We apply our changes and overwrite the original record with our modified record.

It is also possible to perform an update using `setData()`, just as we would do for a non-SQL model. The model indexes that we retrieve are for a given row and column:

```
model.select();
if (model.rowCount() == 1) {
    model.setData(model.index(0, 1), "Melody A.M.");
    model.setData(model.index(0, 3),
                  model.data(model.index(0, 3)).toInt() + 1);
    model.submitAll();
}
```

Deleting a record is similar to updating:

```
model.setTable("cd");
model.setFilter("id = 125");
model.select();
if (model.rowCount() == 1) {
    model.removeRows(0, 1);
    model.submitAll();
}
```

The `removeRows()` call takes the row number of the first record to delete and the number of records to delete. The next example deletes all the records that match the filter:

```
model.setTable("cd");
model.setFilter("year < 1990");
model.select();
if (model.rowCount() > 0) {
    model.removeRows(0, model.rowCount());
    model.submitAll();
}
```

The `QSqlQuery` and `QSqlTableModel` classes provide an interface between Qt and an SQL database. Using these classes, we can create forms that present data to users and that let them insert, update, and delete records.

For projects that use the SQL classes, we must add the line

```
QT += sql
```

to their `.pro` file. This will ensure that the application is linked against the *QtSql* library.

Page 1 of 4 Next >

Discussions

Make a New Comment