Chapter 9

The QtSql Module

Nowadays, it is difficult to imagine many applications being able to function without relational databases to back them up. For this reason Qt provides a range of classes in the QtSql module that work with various relational database management systems (DBMS). Relational tables and queries can also be used as the basis of Interview models.

9.1 Structure of the QtSql Module

The QtSql module is an independent library that can load additional plugins if required. In contrast to QtCore and QtGui, its contents are not integrated by default (with qmake -project) into the generated projects. In order to use the library, the following entry is therefore necessary in the .pro file:

```
QT += sql
```

To be able to work with the classes of the module, Qt provides a meta-include for this package as well, which contains all the class definitions from the module. The command to integrate it into a source file is as follows:

```
#include <QtSql>
```

Each of the classes of the module belong to one of three layers. The *driver layer* implements the interface between the drivers for various databases and the *API layer* (see Table 9.1). This provides application developers with access to the databases and enables typical SQL operations, such as browsing or modifying tables or querying data.

In order to include the results of queries in Interview views, the *user interface layer* provides models that are based on SQL tables or queries. Figure 9.1 provides an overview of the layers and the classes belonging to them.

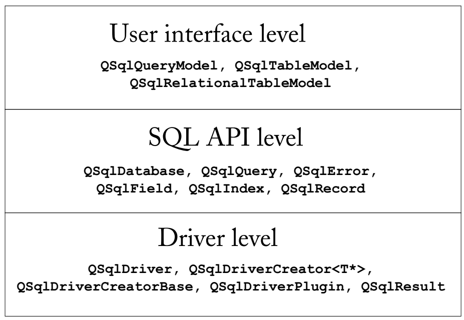

Figure 9.1: The structure of the QtSql module

9.2 Selecting the Appropriate Driver

Since the license of the client API for some database systems is not GPL-compatible, a number of drivers are missing (marked in Table 9.1 with *)) in the open source edition.

Table 9.1: Drivers for QtSql

Driver name	Database system
QDB2	IBM DB2 (Version 7.1 and newer)*)
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle Call Interface driver (versions 8, 9, and 10)*)
QODBC	Open Database Connectivity (ODBC), used by Microsoft SQL server and other ODBC-capable databases

continued

Driver name	Database system
QPSQL	PostgreSQL (version 7.3 and newer)
QSQLITE2	SQLite (version 2)
QSQLITE	SQLite (version 3)
QTDS	Sybase Adaptive Server[*)]

If the Qt version originates from packages of a Linux distribution, you may need to install additional packages. Ubuntu stores the SQL library in the package libqt4-sql, whereas OpenSUSE, in addition to installing qt-sql, requires a DBMS-specific database package, such as qt-sql-mysql for MySQL.

If you build Qt from the sources, you should take a look at the output of ./configure --help:

```
...
-Istring ........... Add an explicit include path.
...
-qt-sql-<driver> ...... Enable a SQL <driver> in the Qt Library, by
                        default none are turned on.
-plugin-sql-<driver> .. Enable SQL <driver> as a plugin to be linked
                        to at run time.
-no-sql-<driver> ...... Disable SQL <driver> entirely.

                        Possible values for <driver>:
                        [ db2 ibase mysql oci odbc psql sqlite
                          sqlite2 tds ]

                        Auto-Detected on this system:
                        [ sqlite ]
...
```

By default Qt builds the driver modules as ***plugins*** for all systems found automatically—in this case for SQLite. If you do not want to compile one of these explicitly, the -no-sql-***driver*** switch is used; for example, in the case of SQLite the switch would be -no-sql-sqlite. Qt also includes its own SQLite version. If you want to use a version of SQLite installed on the system instead, you must also specify the -system-sqlite switch.

If ./configure cannot find an installed database system, despite the development packages installed, then you can specify the include directory of the database system with the -I switch, for example -I/usr/include/mysql, in the case of MySQL. It is left to each user to decide whether a driver is built separately as a plugin (-plugin-sql-***driver***) or compiled permanently into the library (-qt-sql-***driver***). Plugins are more flexible, whereas compiled-in drivers are simpler to handle if the Qt library is to be included in the program.

9.3 Making a Connection

The QSqlDatabase class is used to manage contact with the database server, and its addDatabase() static method returns an instance of QSqlDatabase:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL");
```

As an argument, addDatabase() expects at least the name of the database driver in string form, thus something like "QPSQL" for the Postgres driver. A QSqlDatabase instance generated in this manner serves as the standard connection. If the program needs to establish contact with more than one database, the addDatabase() method additionally requires a connection name:

```
QSqlDatabase webdb =
        QSqlDatabase::addDatabase("QMYSQL", "WebServerDB");
QSqlDatabase personaldb =
        QSqlDatabase::addDatabase("QOCI", "PersonalDB");
QSqlDatabase embeddeddb =
        QSqlDatabase::addDatabase("QSQLITE", "EmbeddedDB");
```

If this argument had been omitted in the variable definitions above, all three QSqlDatabase instances would end up pointing to the SQLite database, since each addDatabase() call without additional parameters modifies the standard connection.

In the following example we set up a connection to a single MySQL server. We establish a connection to a database on this server using a QSqlDatabase object initialized with the relevant driver. To do this we declare the server name, the name of the database, the username, and the password:

```
// sqlexample/main.cpp

#include <QtGui>
#include <QtSql>
#include <QDebug>

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

  QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
  db.setHostName("datenbankserver.example.com");
  db.setDatabaseName("firma");
  db.setUserName("user");
  db.setPassword("pass");

  if (!db.open()) {
     qDebug() << db.lastError();
     return 1;
  }
```

The open() method establishes the connection to the database with this access data. Whether the attempt to connect was successful or not is indicated by its Boolean return value. In case of error, we can determine the reason for the connection failure by using lastError(). The method returns an object of type QSqlError, which qDebug() can read out. If you want to reuse this error object elsewhere, the QSqlError class method text() can be used.

9.4 Making Queries

In the following examples we will work with two tables: The employees table holds information on the employees in a company (Table 9.2), and the departments table (Table 9.3) describes the various organizational units in the company.

id	last name	first name	department
1	Werner	Max	1
2	Lehmann	Daniel	2
3	Roetzel	David	1
4	Scherfgen	David	2
5	Scheidweiler	Najda	2
6	Jueppner	Daniela	4
7	Hasse	Peter	4
8	Siebigteroth	Jennifer	3

Table 9.2: The employees *table from the example database*

id	name
1	Management
2	Development
3	Marketing
4	Accounting

Table 9.3: The departments *table from the example database*

For database operations we use the QSqlQuery class. If a class used in the constructor is given an SQL command as a string, the instanced object immediately carries out this statement. You can re-run the command stored in the query object later on using exec() (for example, after modification to the database). If there are several open connections, the QSqlQuery class accepts a QSqlDatabase instance as a second parameter.

If the SQL operation was successful, the QSqlQuery object is regarded as active, which can be checked with isActive(). If it has collected datasets, for example through a SELECT query, you can navigate through them: first() jumps to the first dataset, last() to the last one, next() to the next one, and previous() to the previous one. With seek() you can address a specific dataset by specifying an integer index. The number of datasets contained in the query object is revealed with size().

The QSqlQuery::record() method returns a QSqlRecord object. It contains information on the response to a SELECT query. Using it we can learn, for example, the numerical index of a specified column in the query result via QSqlRecord::indexOf(). We can use this index to read the value in that column of a dataset (row) in the result with QSqlQuery::value(). The row is determined by the current position in the query object, which we can retrieve using QSqlQuery::at() and change using QSqlQuery::next().

```
// sqlexample/main.cpp (continued)


  QSqlQuery query("SELECT firstname, lastname FROM employees");
  QSqlRecord record = query.record();
  while (query.next()) {
    QString  firstname =
       query.value(record.indexOf("firstname")).toString();
    QString  lastname =
       query.value(record.indexOf("lastname")).toString();
    qDebug() << query.at() << ":" << lastname << "," << firstname;
  }
```

For operations that change the contents of the database (such as UPDATE or DELETE), numRowsAffected() returns the number of datasets involved:

```
// sqlexample/main.cpp (continued)

  query.exec("DELETE FROM employees WHERE lastname = 'Hasse'");
  qDebug() << query.numRowsAffected(); // "1"
```

Things are a little more complicated for INSERT instructions. Since these are used to write values from the program's own data structures to the database, it can be quite complicated to construct a string containing the corresponding SQL instruction. For this reason we take a different path: Using prepare() we save a template for the desired command, equipped with placeholders, in the QSqlQuery object:

```
// sqlexample/main.cpp (continued)

  query.prepare("INSERT INTO employees (lastname, firstname, department)"
                "VALUES(:lastname, :firstname, :department)");
  query.bindValue(":lastname", "Hasse");
  query.bindValue(":firstname", "Peter");
  query.bindValue(":department", 3);
  query.exec();
```

The ***named wildcards*** in the VALUES part of the SQL command, originating from the Oracle world, each begin with a colon. Using the bindValue() command we can replace them with the specific values.

QSqlQuery can also handle the ***unknown parameters*** familiar from the ODBC using addBindValue() . Each call to this method replaces one of the question marks in the VALUES clause, in the order in which they appear:

```
// sqlexample/main.cpp (continued)

  query.prepare("INSERT INTO employees (lastname, firstname, department)"
                "VALUES(?, ?, ?)");
  query.addBindValue("Schwan");
  query.addBindValue("Waldemar");
  query.addBindValue(3);
  query.exec();
```

If you don't want to specify the unknown values according to the sequence of occurrence, you can use the following overloaded variant:

```
  query.bindValue(2, 3);
  query.bindValue(0, "Schwan");
  query.bindValue(1, "Waldemar");
```

Here the first parameter specifies the position of the question mark to be replaced in the prepare() string.

bindValue() also plays a central role in the use of ***stored procedures***, because the parameters of these procedures can be declared both as IN and as OUT. Parameters declared as cmdOUT function as return values.

In order to access a return value, we must adjust the bindValue() method: The value passed does not matter here, as it will be overwritten by the OUT value later. But the QSql::Out specification, which tells QSqlQuery to overwrite the value, is important here. After we have executed exec(), the value lies at the corresponding position. We can check this with boundValue():

```
// sqlexample/main.cpp (continued)

  query.prepare("CALL countEmployees(?)");
  query.bindValue(0, 0, QSql::Out);
  query.exec();
  qDebug() << query.boundValue(0).toInt()
```

Unfortunately, this approach does not work correctly in MySQL 5, due to API limitations. In order to access the OUT values under MySQL 5, we must make two queries manually: First we run the stored procedure with CALL, and then we read in the value produced, using SELECT. In order to refer to the value, in each case we use a wildcard with @ as a MySQL-specific prefix, so that we can read out the return value of the stored procedure as a dataset:

```
// sqlexample/main.cpp (continued)

  query.exec("CALL countEmployees(@outwert)");
  query.exec("SELECT @outwert");
  query.next();
  qDebug() << query.value(0);
  return 0;
}
```

9.5 Transactions

Not all database systems support transactions, which combine several SQL operations into an atomic operation. To help the Qt programmer keep the code portable, QSqlDriver can therefore be asked about its transaction capabilities with hasFeature():

```
if (db.driver()->hasFeature(QSqlDriver::Transactions)) ... ;
```

If the driver supports transactions, you can introduce them with the QSqlDatabase method transaction(). If all operations are completed, the transaction is closed with commit(). If an error occurs, rollback() undoes all the operations of the current transaction.

9.6 Embedded Databases

Qt's SQLite driver enables data to be stored in a relational database and queried, without an external database server. There are restrictions, of course, but the demands made of embedded databases are usually less severe than those for databases residing on dedicated servers, and SQLite is intended for just such situations. This means that SQLite cannot handle stored procedures and does not scale as well as its big brothers. It is well suited, however, to applications that need a basic relational data store. A perfect example is the KDE music player Amarok, which stores metadata about pieces of music in a SQLite database.

To open a connection to a SQLite database, you only need to specify a database name. The SQLite driver expects a filename in this case:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName("firma.db");
```

If the database should only remain in memory while the program is running, a temporary database can be generated by enclosing the database name within colons, as shown below. The :results: database will not not be saved as a file when the program terminates:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName(":results:");
```

You can work with this database as normal, with the understanding that any changes made to it will later be lost. A temporary database does not need its own data structures if the data is already of a relational nature.

9.7 Using SQL Model Classes with Interview

In order to display the contents of databases, table views are usually appropriate, and in some cases, list views are as well. This is why the QtSql module has a range of models for Interview (see Chapter 8 on page 207).

9.7.1 Displaying SQL Tables Without Foreign Keys in Table and Tree Views

QSqlTableModel enables complete tables to be displayed directly in a table or tree view. The column headers correspond to the field names (attributes, columns) of the SQL table. In our personnel database from Table 9.2 on page 261, these are id, first name, last name, and department. Each line corresponds to a dataset. To illustrate this better, we will look at the following example, which assumes an open standard connection:

```
// sqlmvd/main.cpp

...
  QTableView tableView;
  QSqlTableModel tableModel;
  tableModel.setTable("employees");
  tableModel.select();
  tableModel.removeColumn(0);
  tableView.setModel(&tableModel);
  tableView.setWindowTitle("'employees' table");
  tableView.show();
```

First we create a table view and then the model. We allocate a table to it from the current database and order it to fetch data with select(). Then we remove the id column from the view using removeColumn() (Figure 9.2). This method originates from QAbstractItemModel, the ultimate base class of all models. Finally, we bind the model to the table view, give the table a name, and display it.

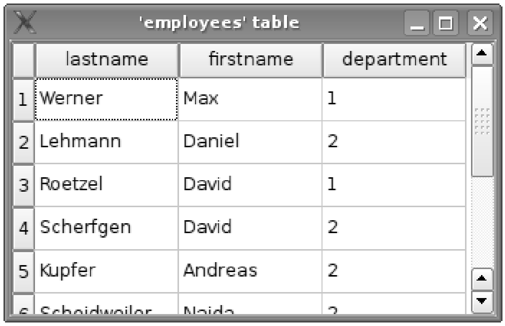

Figure 9.2: QTableModel *is responsible for SQL tables in Interview.*

9.7.2 Resolving Foreign Key Relations

QSqlRelationalTableModel extends the functionality of the table model for use with relational databases. In addition, objects in this class set off foreign key relations. We can use these to replace the uninformative number shown in the department field with the name of the department, by making use of the departments table (see Table 9.3 on page 261).

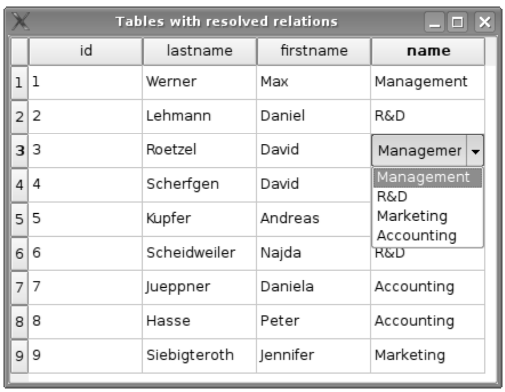

Figure 9.3: QSqlRelationalTableModel *records the foreign key field* id *with the help of a second table.*

To describe this relation, the setRelation() method is used: It expects the number of the column containing the foreign key as the first argument. In our example, the value in the name field from the departments table should appear in the third column instead of the value in the foreign key field (that is, the id field) of the departments table. This information is encapsulated in the instance rel of the QSqlRelation help class, which we pass to setRelation() as the second argument.

Now we can start the query via select(), bind the model to the view, and display the results, as in the previous example:

```
// sqlmvd/main.cpp (continued)

  QTableView tableRelationalView;
  QSqlRelationalTableModel tableRelationalModel;
  tableRelationalModel.setTable("employees");
  QSqlRelation rel("departments", "id", "name");
  tableRelationalModel.setRelation(3, rel);
  tableRelationalModel.select();
  tableRelationalView.setModel(&tableRelationalModel);
  tableRelationalView.setItemDelegate(
                 new QSqlRelationalDelegate(&tableRelationalView));
  tableRelationalView.setWindowTitle(
                "Tables with resolved relations");
  tableRelationalView.show();
```

This is now followed by a peculiarity that only functions in combination with QRelationalTableModel: A special delegate called QSqlRelationalDelegate allows the user to select the value from a list when editing columns for which a relation is defined (Figure 9.3). It compiles these independently from the QSqlRelation used. In the example it takes suggestions from the name column; the value written back to the table, on the other hand, comes from the id column.

9.7.3 Displaying Query Results

To display the results of particularly complex SELECT queries that cannot simply be modeled on a QSqlTableModel with a filter, make use of the QSqlQueryModel. The following example evaluates how many employees the company has in each department. In addition the columns should bear descriptive names, as can be seen in Figure 9.4.

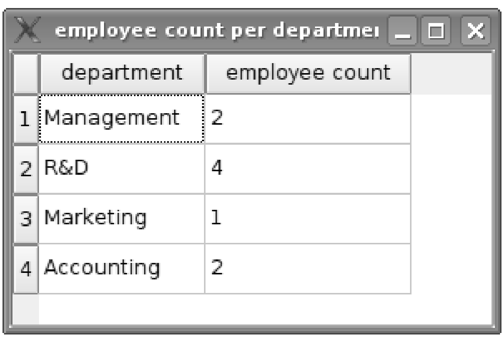

Figure 9.4: QSqlQueryModel *is used as a source for queries of all types in Interview.*

After instantiating the model, we pass the query as a string to setQuery(). Alternatively, we could use a QSqlQuery object.

Since errors can occur in more complex queries, we should introduce an error check immediately after the query executes. lastError() returns the last error announced by the SQL server in an QSqlError object. If this is valid, an error has occurred, which we can display with qDebug():

```
// sqlmvd/main.cpp (continued)

  QTableView queryView;
  QSqlQueryModel queryModel;
  queryModel.setQuery("SELECT departments.name, "
        "COALESCE(COUNT(employees.lastname),0) "
        "FROM departments LEFT JOIN employees "
        "ON employees.department = departments.id "
        "GROUP BY employees.department");

  if (queryModel.lastError().isValid())
    qDebug() << queryModel.lastError();

  queryModel.setHeaderData(0, Qt::Horizontal,
                              QObject::tr("department"));
  queryModel.setHeaderData(1, Qt::Horizontal,
                              QObject::tr("employee count"));
  queryView.setModel(&queryModel);
  queryView.setWindowTitle("employee count per department");
  queryView.show();
```

We can achieve user-friendly column headers by replacing the first two column headers withsetHeaderData().[1] Then we bind the model to the view and display the view, as before, with a customized heading.

9.7.4 Editing Strategies

All of these table models are writable. However, we have not yet looked closely at the point in time when the model writes the data back to the database.

QSqlTableModel and QSqlRelationalTableModel know three *editing strategies*, which are allocated to models using setEditStrategy(). They are as follows:

SqlTableModel::OnRowChange
This is the default in all models. If this strategy is active, the model sends an

[1] This can also be done with the SQL instruction AS, of course, but then you would have to ensure, via tr(), that the query can be internationalized; otherwise, the column headers cannot be transferred to other languages.

UPDATE for the dataset as soon as the user selects another dataset—that is, another row in the view.

SqlTableModel::OnFieldChange
: This transfers every change to the database directly after the user has changed a value in a field.

SqlTableModel::OnManualSubmit
: This temporarily saves all changes in the model until either the submitAll() slot, which transfers all changes to the database, or the revertAll() slot is triggered. The latter rejects all cached data and restores the status from the database (see Chapter 9.7.5 on page 270 for more on the revertAll() slot).

We will illustrate this last scenario by modifying the example from page 265 so that it additionally contains two buttons that are arranged in a layout beneath the table view. All other commands are left as they are.

```
// sqlmvd/main.cpp (continued)

  QWidget w;
  QPushButton *submitPb = new QPushButton(
              QObject::tr("Save Changes"));
  QPushButton *revertPb = new QPushButton(
              QObject::tr("Roll back changes"));
  QGridLayout *lay = new QGridLayout(&w);
  QTableView *manualTableView = new QTableView;
  lay->addWidget(manualTableView, 0, 0, 1, 2);
  lay->addWidget(submitPb, 1, 0);
  lay->addWidget(revertPb, 1, 1);
  QSqlTableModel manualTableModel;
  manualTableModel.setTable("employees");
  manualTableModel.select();
  manualTableModel.setEditStrategy(
              QSqlTableModel::OnManualSubmit);
  manualTableView->setModel(&manualTableModel);
  QObject::connect(submitPb, SIGNAL(clicked(bool)),
              &manualTableModel, SLOT(submitAll()) );
  QObject::connect(revertPb, SIGNAL(clicked(bool)),
              &manualTableModel, SLOT(revertAll()) );
  w.setWindowTitle("manually revertable table");
  w.show();

  return app.exec();
}
```

After converting the editing strategy to OnManualSubmit, we insert two signal/slot connections: A click on the submitPb button calls the submitAll() slot, whereas revertPb triggers revertAll().

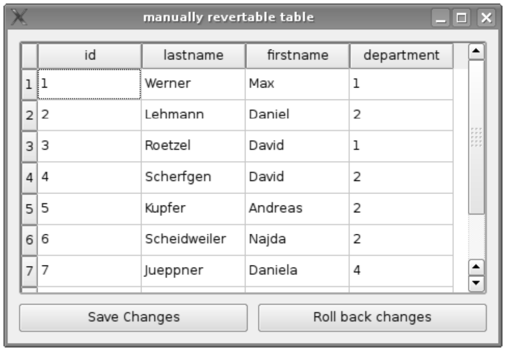

Figure 9.5: With the OnManualSubmit *editing strategy, local changes can be transferred at any time you want to the database.*

Now we must not forget to display the main widget w as the new top-level widget. The result is illustrated in Figure 9.5.

9.7.5 Errors in the Table Model

Several problems that occur in connection with the table models in Qt 4.1 should not be left unaddressed at this point. One is that editor operations do not always function reliably after columns have been removed. The QSqlRelationalTableModel even ignores the removeColumn() instruction entirely. As a workaround, a proxy model that filters out the unwanted datasets is recommended here. If the data should only be displayed, you can instead simply place an SQL query above the QSqlQueryModel.

Another problem involves the revertAll() slot, which is intended to undo all changes in relational tables with the OnManualSubmit editing strategy. However, in the columns in which a foreign key relation was previously defined with setRelation(), revertAll() does not revert back to the old values. The only solution until now was to connect the slot of the button with a custom-developed slot that replaces the current model with a new one that has the same properties. Since the model temporarily saves the data, it will be lost in this way, and the new model will display the original data from the database.