Databases with Qt

Qt 3.0

The definitive Qt documentation is provided in HTML format supplied with Qt, and available online at http://doc.trolltech.com. This PDF file was generated automatically from the HTML source as a convenience to users, although PDF is not an official Qt documentation format.

Contents

SQL Module

This module is part of the Qt Enterprise Edition.

Introduction

Qt's SQL classes help you provide seamless database integration to your Qt applications.

This overview assumes that you have at least a basic knowledge of SQL. You should be able to understand simple `SELECT`, `INSERT`, `UPDATE` and `DELETE` commands. Although the QSqlCursor class provides an interface to database browsing and editing that does not *require* a knowledge of SQL, a basic understanding of SQL is highly recommended. A standard text covering SQL databases is *An Introduction to Database Systems (7th ed.)* by C. J. Date, ISBN 0201385902.

Whilst this module overview presents the classes from a purely programmatic point of view the *Qt Designer* manual's "Creating Database Applications" chapter takes a higher-level approach demonstrating how to set up master-detail relationships between widgets, perform drilldown and handle foreign key lookups.

This document is divided into six sections:

SQL Module Architecture. This describes the how the classes fit together.

Connecting to Databases. This section explains how to set up database connections using the QSqlDatabase class.

Executing SQL Commands. This section demonstrates how to issue the standard data manipulation commands, `SELECT`, `INSERT`, `UPDATE` and `DELETE` on tables in the database (although any valid SQL statement can be sent to the database). The focus is purely on database interaction using QSqlQuery.

Using Cursors. This section explains how to use the QSqlCursor class which provides a more structured and powerful API than the raw SQL used with QSqlQuery.

Data-Aware Widgets. This section shows how to programmatically link your database to the user interface. In this section we introduce the QDataTable, QSqlForm, QSqlPropertyMap and QSqlEditorFactory classes and demonstrate how to use custom data-aware widgets. *Qt Designer* provides an easy visual way of achieving the same thing. See the *Qt Designer* manual, QDataBrowser and QDataView for more information.

Subclassing QSqlCursor. This section gives examples of subclassing QSqlCursor. Subclassing can be used to provide default and calculated values for fields (such as auto-numbered primary index fields), and to display calculated data, e.g. showing names rather than ids of foreign keys.

SQL Module Architecture

The SQL classes are divided into three layers:

User Interface Layer. These classes provide data-aware widgets that can be connected to tables or views in the database (by using a QSqlCursor as a data source). End users can interact directly with these widgets to browse or edit data. *Qt Designer* is fully integrated with the SQL classes and can be used to create data-aware forms. The data-aware widgets can also be programmed directly with your own C++ code. The classes that support this layer

include QSqlEditorFactory, QSqlForm, QSqlPropertyMap, QDataTable, QDataBrowser and QDataView.

SQL API Layer. These classes provide access to databases. Connections are made using the QSqlDatabase class. Database interaction is achieved either by using the QSqlQuery class and executing SQL commands directly or by using the higher level QSqlCursor class which composes SQL commands automatically. In addition to QSqlDatabase, QSqlCursor and QSqlQuery, the SQL API layer is supported by QSqlError, QSqlField, QSqlIndex and QSqlRecord.

Driver Layer. This comprises three classes, QSqlResult, QSqlDriver and QSqlDriverFactoryInterface. This layer provides the low level bridge between the database and the SQL classes. This layer is documented separately since it is only relevant to driver writers, and is rarely used in standard database application programming. See here for more information on implementing a Qt SQL driver plugin.

SQL Driver Plugins

The Qt SQL module can dynamically load new drivers at runtime using the Plugins.

The SQL driver documentation describes how to build plugins for specific database management systems.

Once a plugin is built, Qt will automatically load it, and the driver will be available for use by QSqlDatabase (see QSqlDatabase::drivers() for more information).

Connecting to Databases

At least one database connection must be created and opened before the QSqlQuery or QSqlCursor classes can be used.

If the application only needs a single database connection, the QSqlDatabase class can create a connection which is used by default for all SQL operations. If multiple database connections are required these can easily be set up.

QSqlDatabase requires the qsqldatabase.h header file.

Connecting to a Single Database

Making a database connection is a simple three step process: activate the driver, set up the connection information, and open the connection.

```
#include <qapplication.h>
#include <qsqldatabase.h>
#include "../login.h"

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

    QSqlDatabase *defaultDB = QSqlDatabase::addDatabase( DB_SALES_DRIVER );
    if ( defaultDB ) {
        defaultDB->setDatabaseName( DB_SALES_DBNAME );
        defaultDB->setUserName( DB_SALES_USER );
        defaultDB->setPassword( DB_SALES_PASSWD );
        defaultDB->setHostName( DB_SALES_HOST );

        if ( defaultDB->open() ) {
            // Database successfully opened; we can now issue SQL commands.
```

```
            }
        }

        return 0;
    }
```

From sql/overview/connect1/main.cpp

First we activate the driver by calling QSqlDatabase::addDatabase(), passing the name of the driver we wish to use for this connection. At the time of writing the available drivers are: QODBC3 (Open Database Connectivity), QOCI8 (Oracle), QTDS7 (Sybase Adaptive Server and Microsoft SQL Server), QPSQL7 (PostgreSQL 6 and 7) and QMYSQL3 (MySQL). Note that some of these drivers aren't included in the Qt Free Edition, look at the README files for details.

The connection which is created becomes the application's default database connection and will be used by the Qt SQL classes if no other database is specified.

Second we call setDatabaseName(), setUserName(), setPassword() and setHostName() to initialize the connection information. Note that for the QOCI8 (Oracle) driver the TNS Service Name has to be passed to setDatbaseName().

Third we call open() to open the database and give us access to the data. If this call fails it will return FALSE; error information can be determined using QSqlDatabase::lastError().

Connecting to Multiple Databases

Connecting to multiple databases is achieved using the two argument form of QSqlDatabase::addDatabase() where the second argument is a unique identifier distinguishing the connection.

```
#include <qapplication.h>
#include <qsqldatabase.h>
#include "../login.h"

bool createConnections();

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

    if ( createConnections() ) {
        // Databases successfully opened; get pointers to them:
        QSqlDatabase *oracledb = QSqlDatabase::database( "ORACLE" );
        // Now we can now issue SQL commands to the oracle connection
        // or to the default connection
    }

    return 0;
}

bool createConnections()
{
    // create the default database connection
    QSqlDatabase *defaultDB = QSqlDatabase::addDatabase( DB_SALES_DRIVER );
    if ( ! defaultDB ) {
        qWarning( "Failed to connect to driver" );
        return FALSE;
    }
    defaultDB->setDatabaseName( DB_SALES_DBNAME );
    defaultDB->setUserName( DB_SALES_USER );
```

```
        defaultDB->setPassword( DB_SALES_PASSWD );
        defaultDB->setHostName( DB_SALES_HOST );
        if ( ! defaultDB->open() ) {
            qWarning( "Failed to open sales database: " +
                      defaultDB->lastError().driverText() );
            qWarning( defaultDB->lastError().databaseText() );
            return FALSE;
        }

        // create a named connection to oracle
        QSqlDatabase *oracle = QSqlDatabase::addDatabase( DB_ORDERS_DRIVER, "ORACLE" );
        if ( ! oracle ) {
            qWarning( "Failed to connect to oracle driver" );
            return FALSE;
        }
        oracle->setDatabaseName( DB_ORDERS_DBNAME );
        oracle->setUserName( DB_ORDERS_USER );
        oracle->setPassword( DB_ORDERS_PASSWD );
        oracle->setHostName( DB_ORDERS_HOST );
        if ( ! oracle->open() ) {
            qWarning( "Failed to open orders database: " +
                      oracle->lastError().driverText() );
            qWarning( oracle->lastError().databaseText() );
            return FALSE;
        }

        return TRUE;
    }
```

From sql/overview/create_connections/main.cpp

In the example above we have moved the connections into their own function, createConnections(), and added some basic error handling. The static function QSqlDatabase::database() can be called from anywhere to provide a pointer to a database connection. If we call it without any parameter it will return the default connection. If called with the identifier we've used for a connection, e.g. "ORACLE", in the above example, it will return a pointer to the specified connection.

If you create a `main.cpp` using *Qt Designer*, it will *not* include our example createConnections() function. This means that applications that preview correctly in *Qt Designer* will not run unless you implement your own database connections function.

Note that in the code above the ODBC connection was not named and is therefore used as the default connection. QSqlDatabase maintains ownership of the pointers returned by the addDatabase() static function. To remove a database from the list of maintained connections, first close the database with QSqlDatabase::close(), and then remove it using the static function QSqlDatabase::removeDatabase().

Executing SQL Commands Using QSqlQuery

The QSqlQuery class provides an interface for executing SQL commands. It also has functions for navigating through the result sets of `SELECT` queries and for retrieving individual records and field values.

The QSqlCursor class described in the next section inherits from QSqlQuery and provides a higher level interface that composes SQL commands for us. QSqlCursor is particularly easy to integrate with on-screen widgets. Programmers unfamiliar with SQL can safely skip this section and use the QSqlCursor class covered in "Using QSqlCursor".

Transactions

If the underlying database engine supports transactions QSqlDriver::hasFeature(QSqlDriver::Transactions) will return TRUE. You can use QSqlDatabase::transaction() to initiate a transaction, followed by the SQL commands you want to execute within the context of the transaction, and then either QSqlDatabase::commit() or QSqlDatabase::rollback().

Basic Browsing

```
#include <qapplication.h>
#include <qsqldatabase.h>
#include <qsqlquery.h>
#include "../login.h"

bool createConnections();

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

    if ( createConnections() ) {
        QSqlDatabase *oracledb = QSqlDatabase::database( "ORACLE" );
        // Copy data from the oracle database to the ODBC (default)
        // database
        QSqlQuery target;
        QSqlQuery query( "SELECT id, name FROM people;", oracledb );
        if ( query.isActive() ) {
            while ( query.next() ) {
                target.exec( "INSERT INTO people ( id, name ) VALUES ( " +
                             query.value(0).toString() +
                             ", '" + query.value(1).toString() +  "' );" );
            }
        }
    }

    return 0;
}
```

From sql/overview/basicbrowsing/main.cpp

In the example above we've added an additional header file, qsqlquery.h. The first query we create, `target`, uses the default database and is initially empty. For the second query, `q`, we specify the "ORACLE" database that we want to retrieve records from. Both the database connections were set up in the createConnections() function we wrote earlier.

After creating the initial `SELECT` statement, isActive() is checked to see if the query executed successfully. The next() function is used to iterate through the query results. The value() function returns the contents of fields as QVariants. The insertions are achieved by creating and executing queries against the default database using the `target` QSqlQuery.

```
            int count = 0;
            if ( query.isActive() ) {
                while ( query.next() ) {
                    target.exec( "INSERT INTO people ( id, name ) VALUES ( " +
                                 query.value(0).toString() +
                                 ", '" + query.value(1).toString() +  "' );" );
                    if ( target.isActive() )
```

```
                    count += target.numRowsAffected();
                }
            }
```

From sql/overview/basicbrowsing2/main.cpp

The above code introduces a count of how many records are successfully inserted. Note that isActive() returns FALSE if the query, e.g. the insertion, fails. numRowsAffected() returns -1 if the number of rows cannot be determined, e.g. if the query fails.

Basic Data Manipulation

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

    int rows = 0;

    if ( createConnections() ) {
        QSqlQuery query( "INSERT INTO staff ( id, forename, surname, salary ) "
                    "VALUES ( 1155, 'Ginger', 'Davis', 50000 );" );
        if ( query.isActive() ) rows += query.numRowsAffected() ;

        query.exec( "UPDATE staff SET salary=60000 WHERE id=1155;" );
        if ( query.isActive() ) rows += query.numRowsAffected() ;

        query.exec( "DELETE FROM staff WHERE id=1155;" );
        if ( query.isActive() ) rows += query.numRowsAffected() ;
    }

    return ( rows == 3 ) ? 0 : 1;
}
```

From sql/overview/basicdatamanip/main.cpp

This example demonstrates straightforward SQL DML (data manipulation language) commands. Since we did not specify a database in the QSqlQuery constructor the default database is used. QSqlQuery objects can also be used to execute SQL DDL (data definition language) commands such as `CREATE` TABLE and `CREATE` INDEX.

Navigating Result Sets

Once a `SELECT` query has been executed successfully we have access to the result set of records that matched the query criteria. We have already used one of the navigation functions, next(), which can be used alone to step sequentially through the records. QSqlQuery also provides first(), last(), next() and prev(). After any of these commands we can check that we are on a valid record location by calling isValid().

We can also navigate to any arbitrary record using seek(). The first record in the dataset is zero. The number of the last record is size() - 1. Note that not all databases provide the size of a `SELECT` query and in such cases size() returns -1.

```
    if ( createConnections() ) {
        QSqlQuery query( "SELECT id, name FROM people ORDER BY name;" );
        if ( ! query.isActive() ) return 1; // Query failed
        int i;
        i = query.size();              // In this example we have 9 records; i == 9.
        query.first();                 // Moves to the first record.
```

```
        i = query.at();                 // i == 0
        query.last();                   // Moves to the last record.
        i = query.at();                 // i == 8
        query.seek( query.size() / 2 ); // Moves to the middle record.
        i = query.at();                 // i == 4
    }
```

From sql/overview/navigating/main.cpp

The example above shows some of the navigation functions in use.

Not all drivers support size(), but we can interrogate the driver to find out:

```
QSqlDatabase* defaultDB = QSqlDatabase::database();
if ( defaultDB->driver()->hasFeature( QSqlDriver::QuerySize ) ) {
    // QSqlQuery::size() supported
}
else {
    // QSqlQuery::size() cannot be relied upon
}
```

Once we have located the record we are interested in we may wish to retrieve data from it.

```
    if ( createConnections() ) {
        QSqlQuery query( "SELECT id, surname FROM staff;" );
        if ( query.isActive() ) {
            while ( query.next() ) {
                qDebug( query.value(0).toString() + ": " +
                        query.value(1).toString() );
            }
        }
    }
```

From sql/overview/retrieve1/main.cpp

Note that if you wish to iterate through the record set in order the only navigation function you need is next().

Tip: The lastQuery() function returns the text of the last query executed. This can be useful to check that the query you think is being executed is the one actually being executed.

Using QSqlCursor

The QSqlCursor class provides a high level interface to browsing and editing records in SQL database tables or views without the need to write your own SQL.

QSqlCursor can do almost everything that QSqlQuery can, with two exceptions. Since cursors represent tables or views within the database, by default, QSqlCursor objects retrieve all the fields of each record in the table or view whenever navigating to a new record. If only some fields are relevant simply confine your processing to those and ignore the others. Or, manually disable the generation of certain fields using QSqlRecord::setGenerated(). However, if you really don't want to retrieve all fields in the cursor then you should use a QSqlQuery instead, and customize the query to suit your needs. You can edit records using a QSqlCursor providing that the table or view has a primary index that uniquely distinguishes each record. If this condition is not met then you'll need to use a QSqlQuery for edits. (Note that not all databases support editable views.)

QSqlCursor operates on a single record at a time. Whenever performing an insert, update or delete using QSqlCursor, only a single record in the database is affected. When navigating through records in the cursor, only one record at a time is available in application code. In addition, QSqlCursor maintains a separate 'edit buffer' which is

used to make changes to a single record in the database. The edit buffer is maintained in a separate memory area, and is unnaffected by the 'navigation buffer' which changes as the cursor moves from record to record.

Before we can use QSqlCursor objects we must first create and open a database connection. Connecting is described in the Connecting to Databases section above. For the examples that follow we will assume that the connections have been created using the createConnections() function defined in the QSqlDatabase example presented earlier.

In the data-aware widgets section that follows this one we show how to link widgets to database cursors. Once we have a knowledge of both cursors and data-aware widgets we can discuss subclassing QSqlCursor.

The QSqlCursor class requires the qsqlcursor.h header file.

Retrieving Records

```
#include <qapplication.h>
#include <qsqldatabase.h>
#include <qsqlcursor.h>
#include "../login.h"

bool createConnections();

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

    if ( createConnections() ) {
        QSqlCursor cur( "staff" ); // Specify the table/view name
        cur.select(); // We'll retrieve every record
        while ( cur.next() ) {
            qDebug( cur.value( "id" ).toString() + ": " +
                    cur.value( "surname" ).toString() + " " +
                    cur.value( "salary" ).toString() );
        }
    }

    return 0;
}
```

From sql/overview/retrieve2/main.cpp

We create the QSqlCursor object, specifying the table or view to use. If we need to use a database other than the default we can specify it in the QSqlCursor constructor.

The SQL executed by the cur.select() call is

```
SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid FROM staff
```

Next, we iterate through the records returned by this select statement using cur.next(). Field values are retrieved in in a similar way to QSqlQuery, except that we pass field names rather than numeric indexes to value() and setValue().

Sorting and Filtering Records

To specify a subset of records to retrieve we can pass filtering criteria to the select() function. Each record that is returned will meet the criteria of the filter (the filter corresponds to the SQL statement's WHERE clause).

```
cur.select( "id > 100" );
```

This select() call will execute the SQL

```
SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid
FROM staff WHERE staff.id > 100
```

This will retrieve only those staff whose `id` is greater than 100.

In addition to retrieving selected records we often want to specify a sort order for the returned records. This is achieved by creating a QSqlIndex object which contains the names of the field(s) we wish to sort by and pass this object to the select() call.

```
QSqlCursor cur( "staff" );
QSqlIndex nameIndex = cur.index( "surname" );
cur.select( nameIndex );
```

Here we create a QSqlIndex object with one field, "surname". When we call the select() function we pass the index object, which specifies that the records should be returned sorted by staff.surname. Each field in the index object is used in the ORDER BY clause of the select statement. The SQL executed here is

```
SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid
FROM staff ORDER BY staff.surname ASC
```

Combining the retrieval of a subset of records and ordering the results is straightforward.

```
cur.select( "surname LIKE 'A%'", nameIndex );
```

We pass in a filter string (the WHERE clause), and the QSqlIndex object to sort by (the ORDER BY clause). This produces

```
SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid
FROM staff WHERE staff.surname LIKE 'A%' ORDER BY staff.surname ASC
```

To sort by more than one field, an index can be created which contains multiple fields. Ascending and descending order can be set using QSqlIndex::setDescending(); the default is ascending.

```
QSqlCursor cur( "staff" );
QStringList fields = QStringList() << "surname" << "forename";
QSqlIndex order = cur.index( fields );
cur.select( order );
while ( cur.next() ) {
```

From sql/overview/order1/main.cpp

Here we create a string list containing the fields we wish to sort by, in the order they are to be used. Then we create a QSqlIndex object based on these fields, finally executing the select() call using this index. This executes

```
SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid
FROM staff ORDER BY staff.surname ASC, staff.forename ASC
```

If we need to retrieve records with fields that match specific criteria we can create a filter based on an index.

```
QSqlCursor cur( "staff" );
QStringList fields = QStringList() << "id" << "forename";
QSqlIndex order = cur.index( fields );
QSqlIndex filter = cur.index( "surname" );
cur.setValue( "surname", "Bloggs" );
cur.select( filter, order );
while ( cur.next() ) {
```

From sql/overview/order2/main.cpp

This executes

```
SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid
FROM staff WHERE staff.surname='Bloggs' ORDER BY staff.id ASC, staff.forename ASC
```

The "order" QSqlIndex contains two fields, "id" and "forename" which are used to order the results. The "filter" QSqlIndex contains a single field, "surname". When an index is passed as a filter to the select() function, for each field in the filter, a *fieldname=value* subclause is created where the value is taken from the current cursor's value for that field. We use setValue() to ensure the value used is the one we want.

Extracting Data

```
QSqlCursor cur( "creditors" );
QStringList orderFields = QStringList() << "surname" << "forename";
QSqlIndex order = cur.index( orderFields );

QStringList filterFields = QStringList() << "surname" << "city";
QSqlIndex filter = cur.index( filterFields );
cur.setValue( "surname", "Chirac" );
cur.setValue( "city", "Paris" );

cur.select( filter, order );

while ( cur.next() ) {
    int id = cur.value( "id" ).toInt();
    QString name = cur.value( "forename" ).toString() + " " +
                   cur.value( "surname" ).toString();
    qDebug( QString::number( id ) + ": " + name );
}
```

From sql/overview/extract/main.cpp

In this example we begin by creating a cursor on the creditors table. We create two QSqlIndex objects. The first, "order", is created from the "orderFields" string list. The second, "filter", is created from the "filterFields" string list. We set the values of the two fields used in the filter, "surname" and "city", to the values we're interested in. Now we call select() which generates and executes the following SQL:

```
SELECT creditors.city, creditors.surname, creditors.forename, creditors.id
FROM creditors
WHERE creditors.surname = 'Chirac' AND creditors.city = 'Paris'
ORDER BY creditors.surname ASC, creditors.forename ASC
```

The filter fields are used in the WHERE clause. Their values are taken from the cursor's current values for those fields; we set these values ourselves with the setValue() calls. The order fields are used in the ORDER BY clause.

Now we iterate through each matching record (if any). We retrieve the contents of the id, forename and surname fields and pass them on to some processing function, in this example a simple qDebug() call.

Manipulating Records

Records can be inserted, updated or deleted in a table or view using a QSqlCursor providing that the table or view has a primary index that uniquely distinguishes each record. If this is not the case a QSqlQuery must be used instead. (Note that not all databases support editable views.)

Each cursor has an internal 'edit buffer' which is used by all the edit operations (insert, update and delete). The editing process is the same for each operation: acquire a pointer to the relevant buffer; call setValue() to prime the buffer with the values you want; call insert() or update() or del() to perform the desired operation. For example, when inserting a record using a cursor, you call primeInsert() to get a pointer to the edit buffer and then call setValue() on this buffer to set each field's value. Then you call QSQlCursor::insert() to insert the contents of the edit buffer into the database. Similarly, when updating (or deleting) a record, the values of the fields in the edit buffer are used to update (or delete) the record in the database. The 'edit buffer' is unaffected by any cursor navigation functions. Note that if you pass a string value to setValue() any single quotes will be escaped (turned into a pair of single quotes) since a single quote is a special character in SQL.

The primeInsert(), primeUpdate() and primeDelete() methods all return a pointer to the internal edit buffer, however each method can potentially perform different operations on the edit buffer before returning it. By default, QSqlCursor::primeInsert() clears all the field values in the edit buffer (see QSqlRecord::clearValues()). Both QSqlCursor::primeUpdate() and QSqlCursor::primeDelete() initialize the edit buffer with the current contents of the cursor before returning it. All three of these functions are virtual, so you can redefine the behavior (for example, reimplementing primeInsert() to auto-number fields in the edit buffer). Data-aware user-interface controls emit signals, e.g. primeInsert(), that you can connect to; these pass a pointer to the appropriate buffer so subclassing may not be necessary. See subclassing QSqlCursor for more information on subclassing; see the *Qt Designer* manual for more on connecting to the primeInsert() signal.

When insert(), update() or del() is called on a cursor, it will be invalidated and will no longer be positioned on a valid record. If we need to move to another record after performing an insert(), update() or del() we must make a fresh select() call. This ensures that changes to the database are accurately reflected in the cursor.

Inserting Records

```
QSqlCursor cur( "prices" );
QStringList names = QStringList() <<
    "Screwdriver" << "Hammer" << "Wrench" << "Saw";
int id = 20;
for ( QStringList::Iterator name = names.begin();
      name != names.end(); ++name ) {
    QSqlRecord *buffer = cur.primeInsert();
    buffer->setValue( "id", id );
    buffer->setValue( "name", *name );
    buffer->setValue( "price", 100.0 + (double)id );
    count += cur.insert();
    id++;
}
```

From sql/overview/insert/main.cpp

In this example we create a cursor on the "prices" table. Next we create a list of product names which we iterate over. For each iteration we call the cursor's primeInsert() method. This method returns a pointer to a QSqlRecord buffer in which all the fields are set to `NULL`. (Note that QSqlCursor::primeInsert() is virtual, and can be customized by derived classes. See QSqlCursor). Next we call setValue() for each field that requires a value. Finally we call insert() to insert the record. The insert() call returns the number of rows inserted.

We obtained a pointer to a QSqlRecord object from the primeInsert() call. QSqlRecord objects can hold the data for a single record plus some meta-data about the record. In practice most interaction with a QSqlRecord consists of simple value() and setValue() calls as shown in this and the following example.

Updating Records

```
QSqlCursor cur( "prices" );
cur.select( "id=202" );
if ( cur.next() ) {
```

```
        QSqlRecord *buffer = cur.primeUpdate();
        double price = buffer->value( "price" ).toDouble();
        double newprice = price * 1.05;
        buffer->setValue( "price", newprice );
        cur.update();
    }
```

From sql/overview/update/main.cpp

This example begins with the creation of a cursor over the prices table. We select the record we wish to update with the select() call and move to it with the next() call. We call primeUpdate() to get a QSqlRecord pointer to a buffer which is populated with the contents of the current record. We retrieve the value of the price field, calculate a new price, and set the the price field to the newly calculated value. Finally we call update() to update the record. The update() call returns the number of rows updated.

If many identical updates need to be performed, for example increasing the price of every item in the price list, using a single SQL statement with QSqlQuery is more efficient, e.g.

```
QSqlQuery query( "UPDATE prices SET price = price * 1.05"  );
```

Deleting Records

```
    QSqlCursor cur( "prices" );
    cur.select( "id=999" );
    if ( cur.next() ) {
        cur.primeDelete();
        cur.del();
```

From sql/overview/del/main.cpp

To delete records, select the record to be deleted and navigate to it. Then call primeDelete() to populate the cursor with the primary key of the selected record, (in this example, the `prices.id` field), and then call QSqlCursor::del() to delete it.

As with update(), if multiple deletions need to be made with some common criteria it is more efficient to do so using a single SQL statement, e.g.

```
QSqlQuery query( "DELETE FROM prices WHERE id >= 2450 AND id <= 2500" );
```

Data-Aware Widgets

Data-Aware Widgets provide a simple yet powerful means of connecting databases to Qt user interfaces. The easiest way of creating and manipulating data-aware widgets is with *Qt Designer*. For those who prefer a purely programmatic approach the following examples and explanations provide an introduction. Note that the "Creating Database Applications" chapter of the *Qt Designer* manual and its accompanying examples provides additional information.

Data-Aware Tables

```
#include <qapplication.h>
#include <qsqldatabase.h>
#include <qsqlcursor.h>
#include <qdatatable.h>
#include "../login.h"
```

```
bool createConnections();

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

    if ( createConnections() ) {
        QSqlCursor staffCursor( "staff" );
        QDataTable *staffTable = new QDataTable( &staffCursor, TRUE );
        app.setMainWidget( staffTable );
        staffTable->refresh();
        staffTable->show();

        return app.exec();
    }

    return 0;
}
```

From sql/overview/table1/main.cpp

Data-Aware tables require the qdatatable.h and qsqlcursor.h header files. We create our application object, call createConnections() and create the cursor. We create the QDataTable passing it a pointer to the cursor, and set the autoPopulate flag to TRUE. Next we make our QDataTable the main widget and call refresh() to populate it with data and call show() to make it visible.

The autoPopulate flag tells the QDataTable whether or nor it should create columns based on the cursor. autoPopulate does not affect the loading of data into the table; that is achieved by the refresh() function.

```
        QSqlCursor staffCursor( "staff" );
        QDataTable *staffTable = new QDataTable( &staffCursor );

        app.setMainWidget( staffTable );

        staffTable->addColumn( "forename", "Forename" );
        staffTable->addColumn( "surname",  "Surname" );
        staffTable->addColumn( "salary",   "Annual Salary" );

        QStringList order = QStringList() << "surname" << "forename";
        staffTable->setSort( order );

        staffTable->refresh();
        staffTable->show();
```

From sql/overview/table2/main.cpp

We create an empty QDataTable which we make into our main widget and then we manually add the columns we want in the order we wish them to appear. For each column we specify the field name and optionally a display label.

We have also opted to sort the rows in the table; this could also have been achieved by applying the sort to the cursor itself.

Once everything is set up we call refresh() to load the data from the database and show() to make the widget visible.

QDataTables only retrieve visible rows which (depending on the driver) allows even large tables to be displayed very quickly with minimal memory cost.

Creating Data-Aware Forms

Creating data-aware forms is more involved than using data-aware tables because we must take care of each field individually. Most of the code below can be automatically generated by *Qt Designer*. See the *Qt Designer* manual for more details.

Displaying a Record

```
#include <qapplication.h>
#include <qdialog.h>
#include <qlabel.h>
#include <qlayout.h>
#include <qlineedit.h>
#include <qsqldatabase.h>
#include <qsqlcursor.h>
#include <qsqlform.h>
#include "../login.h"

bool createConnections();

class FormDialog : public QDialog
{
    public:
        FormDialog();
};

FormDialog::FormDialog()
{
    QLabel *forenameLabel   = new QLabel( "Forename:", this );
    QLabel *forenameDisplay = new QLabel( this );
    QLabel *surnameLabel    = new QLabel( "Surname:", this );
    QLabel *surnameDisplay  = new QLabel( this );
    QLabel *salaryLabel     = new QLabel( "Salary:", this );
    QLineEdit *salaryEdit   = new QLineEdit( this );

    QGridLayout *grid = new QGridLayout( this );
    grid->addWidget( forenameLabel,     0, 0 );
    grid->addWidget( forenameDisplay,   0, 1 );
    grid->addWidget( surnameLabel,      1, 0 );
    grid->addWidget( surnameDisplay,    1, 1 );
    grid->addWidget( salaryLabel,       2, 0 );
    grid->addWidget( salaryEdit,        2, 1 );
    grid->activate();

    QSqlCursor staffCursor( "staff" );
    staffCursor.select();
    staffCursor.next();

    QSqlForm sqlForm( this );
    sqlForm.setRecord( staffCursor.primeUpdate() );
    sqlForm.insert( forenameDisplay, "forename" );
    sqlForm.insert( surnameDisplay, "surname" );
    sqlForm.insert( salaryEdit, "salary" );
    sqlForm.readFields();
}
```

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

    if ( ! createConnections() ) return 1;

    FormDialog *formDialog = new FormDialog();
    formDialog->show();
    app.setMainWidget( formDialog );

    return app.exec();
}
```

From sql/overview/form1/main.cpp

We include the header files for the widgets that we need. We also include qsqldatabase.h and qsqlcursor.h as usual, but we now add qsqlform.h.

The form will be presented as a dialog so we subclass QDialog with our own FormDialog class. We use a QLineEdit for the salary so that the user can change it. All the widgets are laid out with a grid.

We create a cursor on the staff table, select all records and move to the first record.

Now we create a QSqlForm object and set the QSqlForm's record buffer to the cursor's update buffer. For each widget that we wish to make data-aware we insert a pointer to the widget and the associated field name into the QSqlForm. Finally we call readFields() to populate the widgets with data from the database via the cursor's buffer.

Displaying a Record in a Data Form

QDataView is a Widget that can hold a read-only QSqlForm. In addition to QSqlForm it offers the slot refresh (QSqlRecord *) so it can easily be linked together with a QDataTable to display a detailled view of a record:

```
connect( myDataTable, SIGNAL( currentChanged( QSqlRecord* ) ),
         myDataView, SLOT( refresh( QSqlRecord* ) ) );
```

Editing a Record

This example is similar to the previous one so we will focus on the differences.

```
class FormDialog : public QDialog
{
    Q_OBJECT
    public:
        FormDialog();
        ~FormDialog();
    public slots:
        void save();
    private:
        QSqlCursor staffCursor;
        QSqlForm *sqlForm;
        QSqlIndex idIndex;
};
```

From sql/overview/form2/main.h

The save slot will be used for a button that the user can press to confirm their update. We also hold pointers to the QSqlCursor and the QSqlForm since they will need to be accessed outside the constructor.

```
    staffCursor.setTrimmed( "forename", TRUE );
    staffCursor.setTrimmed( "surname",  TRUE );
```

We call setTrimmed() on the text fields so that any spaces used to right pad the fields are removed when the fields are retrieved.

Properties that we might wish to apply to fields, such as alignment and validation are achieved in the conventional way, for example, by calling QLineEdit::setAlignment and QLineEdit::setValidator.

```
    QLineEdit   *forenameEdit  = new QLineEdit( this );

    QPushButton *saveButton    = new QPushButton( "&Save", this );
    connect( saveButton, SIGNAL(clicked()), this, SLOT(save()) );
```

The FormDialog constructor is similar to the one in the previous example. We have changed the forename and surname widgets to QLineEdits to make them editable and have added a QPushButton the user can click to save their updates.

```
    grid->addWidget( saveButton,    3, 0 );
```

We add an extra row to the grid containing the save button.

```
    idIndex = staffCursor.index( "id" );
    staffCursor.select( idIndex );
    staffCursor.first();
```

We create a QSqlIndex object and then execute a select() using the index. We then move to the first record in the result set.

```
    sqlForm = new QSqlForm( this );
    sqlForm->setRecord( staffCursor.primeUpdate() );
```

We create a new QSqlForm object and set it's record buffer to the cursor's update buffer.

```
    sqlForm->insert( forenameEdit, "forename" );
    sqlForm->insert( surnameEdit, "surname" );
    sqlForm->insert( salaryEdit, "salary" );
    sqlForm->readFields();
```

Now we link the buffer's fields to the QLineEdit controls. (In the previous example we linked the cursor's fields.) The edit controls are populated by the readFields() call as before.

```
FormDialog::~FormDialog()
{

}
```

In the destructor we don't have to worry about the widgets or QSqlForm since they are children of the form and will be deleted by Qt at the right time.

```
void FormDialog::save()
{
    sqlForm->writeFields();
    staffCursor.update();
    staffCursor.select( idIndex );
    staffCursor.first();
}
```

Finally we add the save functionality for when the user presses the save button. We write back the data from the widgets to the QSqlRecord buffer with the writeFields() call. Then we update the database with the updated version of the record with the cursor's update() function. At this point the cursor is no longer positioned at a valid record so we reissue the select() call using our QSqlIndex and move to the first record.

QDataBrowser and QDataView are widgets which provide a great deal of the above functionality. QDataBrowser provides a data form which allows editing of and navigation through records in a cursor. QDataView provides a read only form for data in a cursor or database record. See the class documentation or the *Qt Designer* manual for more information on using these widgets.

Link to sql/overview/form2/main.cpp

Custom Editor Widgets

QSqlForm uses QSqlPropertyMap to handle the transfer of data between widgets and database fields. Custom widgets can also be used in a form by installing a property map that contains information about the properties of the custom widget which should be used to transfer the data.

This example is based on the form2 example in the previous section so we will only cover the differences here. The full source is in sql/overview/custom1/main.h and sql/overview/custom1/main.cpp

```
class CustomEdit : public QLineEdit
{
    Q_OBJECT
    Q_PROPERTY( QString upperLine READ upperLine WRITE setUpperLine )
    public:
        CustomEdit( QWidget *parent=0, const char *name=0 );
        QString upperLine() const;
        void setUpperLine( const QString &line );
    public slots:
        void changed( const QString &line );
    private:
        QString upperLineText;
};
```

We've created a simple subclass of QLineEdit and added a property, upperLineText, which will hold an uppercase version of the text. We also created a slot, changed.

```
        QSqlPropertyMap *propMap;
```

We will be using a property map so we add a pointer to a property map to our FormDialog's private data.

```
CustomEdit::CustomEdit( QWidget *parent, const char *name ) :
    QLineEdit( parent, name )
{
    connect( this, SIGNAL(textChanged(const QString &)),
             this, SLOT(changed(const QString &)) );
}
```

In the CustomEdit constructor we use the QLineEdit constructor and add a connection between the textChanged signal and our own changed slot.

```
void CustomEdit::changed( const QString &line )
{
    setUpperLine( line );
}
```

The changed() slot calls our setUpperLine() function.

```
void CustomEdit::setUpperLine( const QString &line )
{
    upperLineText = line.upper();
    setText( upperLineText );
}
```

The setUpperLine() function places an uppercase copy of the text in the upperLineText buffer and then sets the text of the widget to this text.

Our CustomEdit class ensures that the text entered is always uppercase and provides a property that can be used with a property map to link CustomEdit instances directly to database fields.

```
    CustomEdit  *forenameEdit   = new CustomEdit( this );

    CustomEdit  *surnameEdit    = new CustomEdit( this );
```

We use the same FormDialog as we did before, but this time replace two of the QLineEdit widgets with our own CustomEdit widgets.

Laying out the grid and setting up the cursor is the same as before.

```
    propMap = new QSqlPropertyMap;
    propMap->insert( forenameEdit->className(), "upperLine" );
```

We create a new property map on the heap and register our CustomEdit class and its upperLine property with the property map.

```
    sqlForm = new QSqlForm( this );
    sqlForm->setRecord( staffCursor->primeUpdate() );
    sqlForm->installPropertyMap( propMap );
```

The final change is to install the property map into the QSqlForm once the QSqlForm has been created. This passes responsibility for the property map's memory to QSqlForm which itself is owned by the FormDialog, so Qt will delete them at the right time.

The behaviour of this example is identical to the previous one except that the forename and surname fields will be uppercase since they use our CustomEdit widget.

Custom Editor Widgets for QTables

We must reimpliment QSqlEditorFactory to use custom editor widgets in tables. In the following example we will create a custom editor based on QComboBox and a QSqlEditorFactory subclass to show how a QTable can use a custom editor.

```
class StatusPicker : public QComboBox
{
    Q_OBJECT
    Q_PROPERTY( int statusid READ statusId WRITE setStatusId )
    public:
        StatusPicker( QWidget *parent=0, const char *name=0 );
        int statusId() const;
        void setStatusId( int id );
    private:
        QMap index2id;
};
```

From sql/overview/table3/main.h

We create a property, statusid, and define our READ and WRITE methods for it. The statusid's in the status table will probably be different from the combobox's indexes so we create a QMap to map combobox indexes to/from the statusids that we will list in the combobox.

```
class CustomSqlEditorFactory : public QSqlEditorFactory
{
    Q_OBJECT
    public:
        QWidget *createEditor( QWidget *parent, const QSqlField *field );
};
```

We also need to subclass QSqlEditorFactory declaring a createEditor() function since that is the only function we need to reimplement.

```
StatusPicker::StatusPicker( QWidget *parent, const char *name )
    : QComboBox( parent, name )
{
    QSqlCursor cur( "status" );
    cur.select( cur.index( "name" ) );

    int i = 0;
    while ( cur.next() ) {
        insertItem( cur.value( "name" ).toString(), i );
        index2id[i] = cur.value( "id" ).toInt();
        i++;
    }
```

From sql/overview/table3/main.cpp

In the StatusPicker's constructor we create a cursor over the status table indexed by the name field. We then iterate over each record in the status table inserting each name into the combobox. We store the statusid for each name in the index2id QMap using the same QMap index as the combobox index.

```
int StatusPicker::statusId() const
{
    return index2id[ currentItem() ];
}
```

The statusid property READ function simply involves looking up the combobox's index for the currently selected item in the index2id QMap which maps combobox indexes to statusids.

```
void StatusPicker::setStatusId( int statusid )
{
    QMap::Iterator it;
    for ( it = index2id.begin(); it != index2id.end(); ++it ) {
        if ( it.data() == statusid ) {
            setCurrentItem( it.key() );
            break;
        }
    }
}
```

The statusId() function implements the statusid property's WRITE function. We create an iterator over a QMap and iterate over the index2id QMap. We compare each index2id element's data (statusid) to the id parameter's value.

If we have a match we set the combobox's current item to the index2id element's key (the combobox index), and leave the loop.

When the user edits the status field in the QDataTable they will be presented with a combobox of valid status names taken from the status table. However the status displayed is still the raw statusid. To display the status name when the field isn't being edited requires us to subclass QDataTable and reimplement the paintField() function.

```
class CustomTable : public QDataTable
{
    Q_OBJECT
public:
    CustomTable(
            QSqlCursor *cursor, bool autoPopulate = FALSE,
            QWidget * parent = 0, const char * name = 0 ) :
        QDataTable( cursor, autoPopulate, parent, name ) {}
    void paintField(
            QPainter * p, const QSqlField* field, const QRect & cr, bool );

};
```

From sql/overview/table4/main.h

We simply call the original QDataTable constructor without changing anything. We also declare the paintField function.

```
void CustomTable::paintField( QPainter * p, const QSqlField* field,
                              const QRect & cr, bool b)
{
    if ( !field )
        return;
    if ( field->name() == "statusid" ) {
        QSqlQuery query( "SELECT name FROM status WHERE id=" +
                    field->value().toString() );
        QString text;
        if ( query.next() ) {
            text = query.value( 0 ).toString();
        }
        p->drawText( 2,2, cr.width()-4, cr.height()-4, fieldAlignment( field ), text );
    }
    else {
        QDataTable::paintField( p, field, cr, b) ;
    }
```

From sql/overview/table4/main.cpp

The paintField code is based on QDataTable's source code. We need to make three changes. Firstly add an if clause `field->name() == "statusid"` and look up the textual value for the id with a straighforward QSqlQuery. Secondly call the superclass to handle other fields. The last change is in our main function where we change staffTable from being a QDataTable to being a CustomTable.

Subclassing QSqlCursor

```
#include <qapplication.h>
#include <qsqldatabase.h>
#include <qsqlcursor.h>
#include <qdatatable.h>
```

```
#include "../login.h"

bool createConnections();

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

    if ( createConnections() ) {
        QSqlCursor invoiceItemCursor( "invoiceitem" );

        QDataTable *invoiceItemTable = new QDataTable( &invoiceItemCursor );

        app.setMainWidget( invoiceItemTable );

        invoiceItemTable->addColumn( "pricesid", "PriceID" );
        invoiceItemTable->addColumn( "quantity", "Quantity" );
        invoiceItemTable->addColumn( "paiddate", "Paid" );

        invoiceItemTable->refresh();
        invoiceItemTable->show();

        return app.exec();
    }

    return 1;
}
```

From sql/overview/subclass1/main.cpp

This example is very similar to the table1 example presented earlier. We create a cursor, add the fields and their display labels to a QDataTable, call refresh() to load the data and call show() to show the widget.

Unfortunately this example is unsatisfactory. It is tedious to set the table name and any custom characteristics for the fields every time we need a cursor over this table. And it would be far better if we displayed the name of the product rather than its pricesid. Since we know the price of the product and the quantity we could also show the product cost and the cost of each invoiceitem. Finally it would be useful (or even essential for primary keys) if we could default some of the values when the user adds a new record.

```
class InvoiceItemCursor : public QSqlCursor
{
    public:
        InvoiceItemCursor();
};
```

From sql/overview/subclass2/main.h

We have created a separate header file and subclassed QSqlCursor.

```
InvoiceItemCursor::InvoiceItemCursor() :
    QSqlCursor( "invoiceitem" )
{
    // NOOP
}
```

From sql/overview/subclass2/main.cpp

In our class's constructor we call the QSqlCursor constructor with the name of the table. We don't have any other characteristics to add at this stage.

```
            InvoiceItemCursor invoiceItemCursor;
```

Whenever we require a cursor over the invoiceitem table we can create an InvoiceItemCursor instead of a generic QSqlCursor.

We still need to show the product name rather than the pricesid.

```
        protected:
            QVariant calculateField( const QString & name );
```

From sql/overview/subclass3/main.h

The change in the header file is minimal: we simply add the signature of the calculateField() function since we will be reimplementing it.

```
    InvoiceItemCursor::InvoiceItemCursor() :
        QSqlCursor( "invoiceitem" )
    {
        QSqlFieldInfo productName( "productname", QVariant::String );
        append( productName );
        setCalculated( productName.name(), TRUE );
    }

    QVariant InvoiceItemCursor::calculateField( const QString & name )
    {
        if ( name == "productname" ) {
            QSqlQuery query( "SELECT name FROM prices WHERE id=" +
                        field( "pricesid" )->value().toString() + ";" );
            if ( query.next() )
                return query.value( 0 );
        }

        return QVariant( QString::null );
    }
```

From sql/overview/subclass3/main.cpp

We have changed the InvoiceItemCursor constructor. We now create a new QSqlField called productname and append this to the InvoiceItemCursor's set of fields. We call setCalculated() on productname to identify it as a calculated field. The first argument to setCalculated() is the field name, the second a bool which if TRUE signifies that calculateField() must be called to get the field's value.

```
            invoiceItemTable->addColumn( "productname", "Product" );
```

We add our new fields with addColumn() which adds them to the form and sets their display names.

We have to define our own calculateField() function. In our example database the pricesid in the invoiceitem table is a foreign key into the prices table. We find the name of the product by executing a query on the prices table using the pricesid. This returns the product's name.

We are now able to extend the example to include calculated fields which perform real calculations.

The header file, sql/overview/subclass4/main.h, remains unchanged from the previous example, but the constructor and calculateField() function require some simple expansion. We'll look at each in turn.

```
    InvoiceItemCursor::InvoiceItemCursor() :
        QSqlCursor( "invoiceitem" )
    {
```

```
        QSqlFieldInfo productName( "productname", QVariant::String );
        append( productName );
        setCalculated( productName.name(), TRUE );

        QSqlFieldInfo productPrice( "price", QVariant::Double );
        append( productPrice );
        setCalculated( productPrice.name(), TRUE );

        QSqlFieldInfo productCost( "cost", QVariant::Double );
        append( productCost );
        setCalculated( productCost.name(), TRUE );
    }
```

From sql/overview/subclass4/main.cpp

We create two extra fields, price and cost, and append them to the cursor's set of fields. Both are registered as calculated fields with calls to setCalculated().

```
    QVariant InvoiceItemCursor::calculateField( const QString & name )
    {

        if ( name == "productname" ) {
            QSqlQuery query( "SELECT name FROM prices WHERE id=" +
                        field( "pricesid" )->value().toString() + ";" );
            if ( query.next() )
                return query.value( 0 );
        }
        else if ( name == "price" ) {
            QSqlQuery query( "SELECT price FROM prices WHERE id=" +
                        field( "pricesid" )->value().toString() + ";" );
            if ( query.next() )
                return query.value( 0 );
        }
        else if ( name == "cost" ) {
            QSqlQuery query( "SELECT price FROM prices WHERE id=" +
                        field( "pricesid" )->value().toString() + ";" );
            if ( query.next() )
                return QVariant( query.value( 0 ).toDouble() *
                                 value( "quantity").toDouble() );
        }

        return QVariant( QString::null );
    }
```

From sql/overview/subclass4/main.cpp

The calculateField() function has expanded slightly because now we must calculate the value of three different fields. The productname and price fields are produced by looking up the corresponding values in the prices table keyed by pricesid. The cost field is calculated simply by multiplying the price by the quantity. Note that we cast the cost to a QVariant since that is the type that calculateField() must return.

We've written three separate queries rather than one to make the example more like a real application where it is more likely that each calculated field would be a lookup against a different table or view.

The last feature that we need to add is defaulting values when the user attempts to insert a new record.

```
            QSqlRecord *primeInsert();
```

From sql/overview/subclass5/main.h

We declare our own primeInsert() function since we will need to reimplement this.

The constructor and the calculateField() function remain unchanged.

```
    QSqlRecord *InvoiceItemCursor::primeInsert()
    {
        QSqlRecord *buffer = editBuffer();
        QSqlQuery query( "SELECT NEXTVAL( 'invoiceitem_seq' );" );
        if ( query.next() )
            buffer->setValue( "id", query.value( 0 ) );
        buffer->setValue( "paiddate", QDate::currentDate() );
        buffer->setValue( "quantity", 1 );

        return buffer;
    }
```

From sql/overview/subclass5/main.cpp

We get a pointer to the internal edit buffer that the cursor uses for inserts and updates. The id field is a unique integer that we generate using the invoiceitem_seq. We default the value of the paiddate field to today's date and default the quantity to 1. Finally we return a pointer to the buffer. The rest of the code is unchanged from the previous version.

The Example Tables

The example tables used can be recreated with the following standard SQL. You may need to modify the SQL to match that used by your particular database.

```
create table people (id integer primary key, name char(40))

create table staff (id integer primary key, forename char(40),
                    surname char(40), salary float, statusid integer)

create table status (id integer primary key, name char(30))

create table creditors (id integer primary key, forename char(40),
                        surname char(40), city char(30))

create table prices (id integer primary key, name char(40), price float)

create table invoiceitem (id integer primary key,
                          pricesid integer, quantity integer,
                          paiddate date)
```

A sequence was used in the calculateField() example above. Note that sequences are not supported in all databases.

```
create sequence invoiceitem_seq
```

SQL Module - Drivers

Introduction

The SQL Module uses driver plugins in order to communicate with different database APIs. Since the SQL Module API is database-independent, all database-specific code is contained within these drivers. Several drivers are supplied with Qt and other drivers can be added. The driver source code is supplied and can be used as a model for writing your own drivers.

To build a driver plugin you need the client API that is shipped with every DBMS (Database Management System). Most installation programs also allow you to install "development libraries", and these are what you need. These libraries are responsible for the low-level communication with the DBMS.

The currently available drivers shipped with Qt are:

- QMYSQL3 - MySQL Driver
- QOCI8 - Oracle Call Interface Driver
- QODBC3 - ODBC (Open Database Connectivity) Driver
- QPSQL7 - PostgreSQL v6.x and v7.x Driver
- QTDS7 - Sybase Adaptive Server and Microsoft SQL Server Driver

Note that not all of the plugins are shipped with the Qt Free Edition due to licence incompatibilities with the GPL.

Building the drivers using configure

The Qt configure script automatically detects the available client libraries on your machine. Run "configure -help" to see what drivers may be built. You should get an output similar to this:

```
Possible values for : [ mysql oci odbc psql tds ]
Auto-Detected on this system: [ mysql psql ]
```

Note that configure cannot detect the neccessary libraries and include files if they are not in the standard paths, so it may be necessary to specify these paths using the "-I" and "-L" switches. If your MySQL include files are installed in /usr/local/mysql (or in C:\mysql\include on Windows), then pass the following parameter to configure: "-I/usr/local/mysql" (or "-IC:\mysql\include" for Windows).

Note that on Windows the parameter -I doesn't allow spaces in filenames, so use the 8.3 name instead, i.e. use "C:\progra~1\mysql" instead of "C:\program files\mysql".

Use the `-qt-sql-<driver>` parameter to build the database driver statically into your Qt library or `-plugin-sql-<driver>` to build the driver as a plugin. Look at the chapters below for additional information about required libraries.

Building the plugins manually

QMYSQL3 - MySQL 3.x

General informations

MySQL 3.x doesn't support SQL transactions by default. There are some backends which offer this functionality. Recent versions of the MySQL client libraries (>3.23.34) allow you to use transactions on those modified servers.

If you have a recent client library and connect to a transaction-enabled MySQL server, a call to the QSqlDriver::hasFeature(QSqlDriver::Transactions) function returns TRUE and SQL transactions can be used.

You can find information about MySQL on http://www.mysql.com

How to build the plugin on Unix/Linux

You need the MySQL header files and as well as the shared library "libmysqlclient.so". Depending on your Linux distribution you need to install a package which is usually called "mysql-devel".

Tell qmake where to find the MySQL header files and shared libraries (here it is assumed that MySQL is installed in /usr/local) and run make:

```
cd $QTDIR/plugins/src/sqldrivers/mysql
qmake -o Makefile "INCLUDEPATH+=/usr/local/include" "LIBS+=-L/usr/local/lib -lmysqlclient" mysql.pro
make
```

How to build the plugin on Windows

You need to get the MySQL installation files. Run SETUP.EXE and choose "Custom Install". Install the "Libs & Include Files" Module. Build the plugin as follows (here it is assumed that MySQL is installed in C:\MYSQL):

```
cd %QTDIR%\plugins\src\sqldrivers\mysql
qmake -o Makefile "INCLUDEPATH+=C:\MYSQL\INCLUDE" "LIBS+=C:\MYSQL\LIB\OPT\LIBMYSQL.LIB" mysql.pro
nmake
```

If you are not using a Microsoft compiler, replace "nmake" with "make" in the statement above.

QOCI8 - Oracle Call Interface

How to build the plugin on Unix/Linux

All files required to build driver should ship with the standard Oracle install. For Linux, it may be possible to copy headers from another platform's install.

Oracle library files required to build driver:

- libclntsh.so
- libclntsh.so.8.0
- libwtc8.so

Oracle header files required to build driver:

- nzerror.h

- nzt.h
- oci.h
- oci1.h
- oci8dp.h
- ociap.h
- ociapr.h
- ocidef.h
- ocidem.h
- ocidfn.h
- ociextp.h
- ocikp.h
- ocikpr.h
- odci.h
- oratypes.h
- ori.h
- orid.h
- orl.h
- oro.h
- ort.h

Tell qmake where to find the Oracle header files and shared libraries (here it is assumed that Oracle is installed in /usr/local) and run make:

```
cd $QTDIR/plugins/src/sqldrivers/oci
qmake -o Makefile "INCLUDEPATH+=/usr/local/include" "LIBS+=-L/usr/local/lib -lclntsh -lwtc8" oci.pro
make
```

How to build the plugin on Windows

Choosing the option "Programmer" in the Oracle Client Installer from the Oracle Client Installation CD is sufficient to build the plugin.

Build the plugin as follows (here it is assumed that Oracle Client is installed in C:\oracle):

```
cd %QTDIR%\plugins\src\sqldrivers\oci
qmake -o Makefile "INCLUDEPATH+=C:\oracle\oci\include" oci.pro
nmake
```

If you are not using a Microsoft compiler, replace "nmake" with "make" in the statement above.

QODBC3 - Open Database Connectivity

General informations

ODBC (Open Database Connectivity) is a general interface that allows you to connect to multiple DBMS using a common interface. The QODBC3 driver allows you to connect to an ODBC driver manager and access his datasources. Note that you also need to install and configure ODBC drivers for the ODBC driver manager that is installed on your system. The QODBC3 plugin then allows you to use these data sources in your Qt project.

On Windows systems after 95 an ODBC driver manager should be installed by default, for Unix systems there are some implementations which have to be installed first. Note that every client that uses your application is required to have an ODBC driver manager installed, otherwise the QODBC3 plugin will not work.

The QODBC3 Plugin needs an ODBC compilant driver manager version 2.0 or greater to work. Some ODBC drivers claim to be version 2.0 compilant, but do not offer all needed functionality. The QODBC3 plugin therefore checks whether the data source can be used after a connection has been established and refuses to work if the check fails. If you don't like this behaviour, you can remove the `#define ODBC_CHECK_DRIVER` line from the file `qsql_odbc.cpp`. Do this at your own risk!

How to build the plugin on Unix/Linux

It is recommended that you use unixODBC. You can find the newest version and ODBC drivers at http://www.unixodbc.org. You need the unixODBC header files and shared libraries.

Tell qmake where to find the unixODBC header files and shared libraries (here it is assumed that unixODBC is installed in /usr/local/unixODBC) and run make:

```
cd $QTDIR/plugins/src/sqldrivers/odbc
qmake "INCLUDEPATH+=/usr/local/unixODBC/include" "LIBS+=-L/usr/local/unixODBC/lib -lodbc"
make
```

How to build the plugin on Windows

The ODBC header and include files should already be installed in the right directories. You just have to build the plugin as follows:

```
cd %QTDIR%\plugins\src\sqldrivers\odbc
qmake -o Makefile odbc.pro
nmake
```

If you are not using a Microsoft compiler, replace "nmake" with "make" in the statement above.

QPSQL7 - PostgreSQL version 6 and 7

General information

The QPSQL7 driver supports both version 6 and 7 of PostgreSQL. We recommend compiling the plugin with a recent version of the PostgreSQL Client API (libpq) because it is more stable and still downward compatible.

If you want to link the plugin against the libpq shipped with version 6 we recomment a recent version like PostgreSQL 6.5.3, otherwise a connection to a version 7 server may not work.

The driver auto-detects the server version of PostgreSQL after a connection was successful. If the server is too old or the version information cannot be determined a warning is issued.

For more information about PostgreSQL visit http://www.postgresql.org.

How to build the plugin on Unix/Linux

Just installing "libpq.so" and the corresponding header files is unfortunately not sufficient. You have to get the whole source distribution and run the configure script once (there is no need to build it if you have already installed a binary distribution).

Tell qmake where to find the PostgreSQL header files and shared libraries (here it is assumed that you extracted the PostgreSQL source code in /usr/src/psql and the shared library is installed in /usr/lib) and run make:

```
cd $QTDIR/plugins/src/sqldrivers/psql
qmake -o Makefile "INCLUDEPATH+=/usr/src/psql/src/include /usr/src/psql/src/interfaces/libpq" "LIBS+=-L/us
make
```

QTDS7 - Sybase Adaptive Server and Microsoft SQL Server

How to build the plugin on Unix/Linux

Under Unix, two libraries are available which support the TDS protocol:

- FreeTDS, a free implementation of the TDS protocol (http://www.freetds.org). Note that FreeTDS is not yet stable, so some functionality may not work as expected.

- Sybase Open Client, available from http://www.sybase.com Note for Linux users: Get the Open Client RPM from http://linux.sybase.com

Regardless of which library you use, the shared object file "libsybdb.so" is needed. Set the SYBASE environment variable to point to the directory where you installed the client library and execute qmake:

```
cd $QTDIR/plugins/src/sqldrivers/tds
qmake -o Makefile "INCLUDEPATH=$SYBASE/include" "LIBS=-L$SYBASE/lib -lsybdb"
make
```

How to build the plugin on Windows

You can either use the DB-Library supplied by Microsoft or the Sybase Open Client (http://www.sybase.com). You have to include NTWDBLIB.LIB to build the plugin:

```
cd %QTDIR%\plugins\src\sqldrivers\tds
qmake -o Makefile "LIBS+=NTWDBLIB.LIB" tds.pro
nmake
```

By default the Microsoft library is used on Windows, if you want to force the use of the Sybase Open Client, you have to define Q_USE_SYBASE in %QTDIR%\src\sql\drivers\tds\qsql_tds.cpp.

Troubleshooting

You should always use client libraries that have been compiled with the same compiler as you are using for your project. If you cannot get a source distibution to compile the client libraries yourself, you have to make sure that the pre-compiled library is compatible with your compiler, otherwise you will get a lot of "undefined symbols" errors. Some compilers have tools to convert libraries, e.g. Borland ships the tool `COFF2OMF.EXE` to convert libraries that have been generated with Microsoft Visual C++.

If the compilation of a plugin succeeds but it cannot be loaded, make sure that the following requirements are met:

- Make sure you are using a shared Qt library, you cannot use the plugins with a static build.
- Make sure that the environment variable QTDIR points to the right directory. Go to the $QTDIR/plugins/sqldrivers directory and make sure that the plugin exists in that directory.
- Make sure that the client libraries of the DBMS are available on the system. On Unix, run the command `ldd` and pass the name of the plugin as parameter, for example `ldd libqsqlmysql.so`. You will get a warning if any of the client libraries couldn't be found. On Windows, you can use the dependency walker of Visual Studio.

How to write your own database driver

QSqlDatabase is responsible for loading and managing database driver plugins. When a database is added (see QSqlDatabase::addDatabase()), the appropriate driver plugin is loaded (using QSqlDriverPlugin). QSqlDatabase relies on the driver plugin to provide interfaces for QSqlDriver and QSqlResult.

QSqlDriver is an abstract base class which defines the functionality of a SQL database driver. This includes functions such as QSqlDriver::open() and QSqlDriver::close(). QSqlDriver is responsible for connecting to a database, establish the proper environment, etc. In addition, QSqlDriver can create QSqlQuery objects appropriate for the particular database API. QSqlDatabase forwards many of its function calls directly to QSqlDriver which provides the concrete implementation.

QSqlResult is an abstract base class which defines the functionality of a SQL database query. This includes statements such as SELECT, UPDATE, or ALTER TABLE. QSqlResult contains functions such as QSqlResult::next() and QSqlResult::value(). QSqlResult is responsible for sending queries to the database, returning result data, etc. QSqlQuery forwards many of its function calls directly to QSqlResult which provides the concrete implementation.

QSqlDriver and QSqlResult are closely connected. When implementing a Qt SQL driver, both of these classes must to be subclassed and the abstract virtual methods in each class must be implemented.

To implement a Qt SQL driver as a plugin (so that it is recognized and loaded by the Qt library at runtime), the driver must use the Q_EXPORT_PLUGIN macro. Please read the Qt Plugin documentation for more information on this. You can also check out how this is done in the SQL plugins that is provided with Qt in `QTDIR/plugins/src/sqldrivers` and `QTDIR/src/sql/drivers`.

QDataBrowser Class Reference

The QDataBrowser class provides data manipulation and navigation for data entry forms.

This class is part of the **sql** module.

`#include <qdatabrowser.h>`

Inherits QWidget [Widgets with Qt].

Public Members

- **QDataBrowser** (QWidget * parent = 0, const char * name = 0, WFlags fl = 0)
- **~QDataBrowser** ()
- enum **Boundary** { Unknown, None, BeforeBeginning, Beginning, End, AfterEnd }
- Boundary **boundary** ()
- void **setBoundaryChecking** (bool active)
- bool **boundaryChecking** () const
- void **setSort** (const QSqlIndex & sort)
- void **setSort** (const QStringList & sort)
- QStringList **sort** () const
- void **setFilter** (const QString & filter)
- QString **filter** () const
- virtual void **setSqlCursor** (QSqlCursor * cursor, bool autoDelete = FALSE)
- QSqlCursor * **sqlCursor** () const
- virtual void **setForm** (QSqlForm * form)
- QSqlForm * **form** ()
- virtual void **setConfirmEdits** (bool confirm)
- virtual void **setConfirmInsert** (bool confirm)
- virtual void **setConfirmUpdate** (bool confirm)
- virtual void **setConfirmDelete** (bool confirm)
- virtual void **setConfirmCancels** (bool confirm)
- bool **confirmEdits** () const
- bool **confirmInsert** () const
- bool **confirmUpdate** () const
- bool **confirmDelete** () const
- bool **confirmCancels** () const
- virtual void **setReadOnly** (bool active)
- bool **isReadOnly** () const
- virtual void **setAutoEdit** (bool autoEdit)
- bool **autoEdit** () const
- virtual bool **seek** (int i, bool relative = FALSE)

Public Slots

- virtual void **refresh** ()
- virtual void **insert** ()
- virtual void **update** ()
- virtual void **del** ()
- virtual void **first** ()
- virtual void **last** ()
- virtual void **next** ()
- virtual void **prev** ()
- virtual void **readFields** ()
- virtual void **writeFields** ()
- virtual void **clearValues** ()
- void **updateBoundary** ()

Signals

- void **firstRecordAvailable** (bool available)
- void **lastRecordAvailable** (bool available)
- void **nextRecordAvailable** (bool available)
- void **prevRecordAvailable** (bool available)
- void **currentChanged** (const QSqlRecord * record)
- void **primeInsert** (QSqlRecord * buf)
- void **primeUpdate** (QSqlRecord * buf)
- void **primeDelete** (QSqlRecord * buf)
- void **beforeInsert** (QSqlRecord * buf)
- void **beforeUpdate** (QSqlRecord * buf)
- void **beforeDelete** (QSqlRecord * buf)
- void **cursorChanged** (QSqlCursor::Mode mode)

Properties

- bool **autoEdit** — whether the browser automatically applies edits
- bool **boundaryChecking** — whether boundary checking is active
- bool **confirmCancels** — whether the browser confirms cancel operations
- bool **confirmDelete** — whether the browser confirms deletions
- bool **confirmEdits** — whether the browser confirms edit operations
- bool **confirmInsert** — whether the data browser confirms insertions
- bool **confirmUpdate** — whether the browser confirms updates
- QString **filter** — the data browser's filter
- bool **readOnly** — whether the browser is read-only
- QStringList **sort** — the data browser's sort

Protected Members

- virtual bool **insertCurrent** ()
- virtual bool **updateCurrent** ()
- virtual bool **deleteCurrent** ()
- virtual bool **currentEdited** ()
- virtual QSql::Confirm **confirmEdit** (QSql::Op m)
- virtual QSql::Confirm **confirmCancel** (QSql::Op m)
- virtual void **handleError** (const QSqlError & error)

Detailed Description

The QDataBrowser class provides data manipulation and navigation for data entry forms.

A high-level API is provided to navigate through data records in a cursor, insert, update and delete records, and refresh data in the display.

If you want a read-only form to present database data use QDataView; if you want a table-based presentation of your data use QDataTable.

A QDataBrowser is used to associate a dataset with a form in much the same way as a QDataTable associates a dataset with a table. Once the data browser has been constructed it can be associated with a dataset with setSqlCursor(), and with a form with setForm(). Boundary checking, sorting and filtering can be set with setBoundaryChecking(), setSort() and setFilter(), respectively.

The insertCurrent() function reads the fields from the default form into the default cursor and performs the insert. The updateCurrent() and deleteCurrent() functions perform similarly to update and delete the current record respectively.

The user can be asked to confirm all edits with setConfirmEdits(). For more precise control use setConfirmInsert(), setConfirmUpdate(), setConfirmDelete() and setConfirmCancels(). Use setAutoEdit() to control the behaviour of the form when the user edits a record and then navigates.

The record set is navigated using first(), next(), prev(), last() and seek(). The form's display is updated with refresh(). When navigation takes place the firstRecordAvailable(), lastRecordAvailable(), nextRecordAvailable() and prevRecordAvailable() signals are emitted. When the cursor record is changed due to navigation the cursorChanged() signal is emitted.

If you want finer control of the insert, update and delete processes then you can use the low level functions to perform these operations as described below.

The form is populated with data from the database with readFields(). If the user is allowed to edit, (see setReadOnly()), write the form's data back to the cursor's edit buffer with writeFields(). You can clear the values in the form with clearValues(). Editing is performed as follows:

- *insert* When the data browser enters insertion mode it emits the primeInsert() signal which you can connect to, for example to pre-populate fields. Call writeFields() to write the user's edits to the cursor's edit buffer then call insert() to insert the record into the database. The beforeInsert() signal is emitted just before the cursor's edit buffer is inserted into the database; connect to this for example, to populate fields such as an auto-generated primary key.
- *update* For updates the primeUpdate() signal is emitted when the data browser enters update mode. After calling writeFields() call update() to update the record and connect to the beforeUpdate() signal to manipulate the user's data before the update takes place.
- *delete* For deletion the primeDelete() signal is emitted when the data browser enters deletion mode. After calling writeFields() call del() to delete the record and connect to the beforeDelete() signal, for example to record an audit of the deleted record.

See also Database Classes.

Member Type Documentation

QDataBrowser::Boundary

This enum describes where the data browser is positioned.

The currently defined values are:

- `QDataBrowser::Unknown` - the boundary cannot be determined (usually because there is no default cursor, or the default cursor is not active).
- `QDataBrowser::None` - the browser is not positioned on a boundary, but it is positioned on a record somewhere in the middle.
- `QDataBrowser::BeforeBeginning` - the browser is positioned before the first available record.
- `QDataBrowser::Beginning` - the browser is positioned at the first record.
- `QDataBrowser::End` - the browser is positioned at the last record.
- `QDataBrowser::AfterEnd` - the browser is positioned after the last available record.

Member Function Documentation

QDataBrowser::QDataBrowser (QWidget * parent = 0, const char * name = 0, WFlags fl = 0)

Constructs a data browser which is a child of *parent*, with the name *name* and widget flags set to *fl*.

QDataBrowser::~QDataBrowser ()

Destroys the object and frees any allocated resources.

bool QDataBrowser::autoEdit () const

Returns TRUE if the browser automatically applies edits; otherwise returns FALSE. See the "autoEdit" [p. 44] property for details.

void QDataBrowser::beforeDelete (QSqlRecord * buf) [signal]

This signal is emitted just before the cursor's edit buffer is deleted from the database. The *buf* parameter points to the edit buffer being deleted. You might connect to this signal to capture some auditing information about the deletion.

void QDataBrowser::beforeInsert (QSqlRecord * buf) [signal]

This signal is emitted just before the cursor's edit buffer is inserted into the database. The *buf* parameter points to the edit buffer being inserted. You might connect to this signal to populate a generated primary key for example.

void QDataBrowser::beforeUpdate (QSqlRecord * buf) [signal]

This signal is emitted just before the cursor's edit buffer is updated in the database. The *buf* parameter points to the edit buffer being updated. You might connect to this signal to capture some auditing information about the update.

Boundary QDataBrowser::boundary ()

Returns an enum indicating the boundary status of the browser.

This is achieved by moving the default cursor and checking the position, however the current default form values will not be altered. After checking for the boundary, the cursor is moved back to its former position. See QDataBrowser::Boundary.

See also Boundary [p. 36].

bool QDataBrowser::boundaryChecking () const

Returns TRUE if boundary checking is active; otherwise returns FALSE. See the "boundaryChecking" [p. 44] property for details.

void QDataBrowser::clearValues () [virtual slot]

Clears all the values in the form.

All the edit buffer field values are set to their 'zero state', e.g. 0 for numeric fields and "" for string fields. Then the widgets are updated using the property map. For example, a combobox that is property-mapped to integers would scroll to the first item. See the QSqlPropertyMap constructor for the default mappings of widgets to properties.

QSql::Confirm QDataBrowser::confirmCancel (QSql::Op m) [virtual protected]

Protected virtual function which returns a confirmation for cancelling an edit mode *m*. Derived classes can reimplement this function and provide their own confirmation dialog. The default implementation uses a message box which prompts the user to confirm the edit action.

bool QDataBrowser::confirmCancels () const

Returns TRUE if the browser confirms cancel operations; otherwise returns FALSE. See the "confirmCancels" [p. 44] property for details.

bool QDataBrowser::confirmDelete () const

Returns TRUE if the browser confirms deletions; otherwise returns FALSE. See the "confirmDelete" [p. 44] property for details.

QSql::Confirm QDataBrowser::confirmEdit (QSql::Op m) [virtual protected]

Protected virtual function which returns a confirmation for an edit of mode *m*. Derived classes can reimplement this function and provide their own confirmation dialog. The default implementation uses a message box which prompts the user to confirm the edit action.

bool QDataBrowser::confirmEdits () const

Returns TRUE if the browser confirms edit operations; otherwise returns FALSE. See the "confirmEdits" [p. 44] property for details.

bool QDataBrowser::confirmInsert () const

Returns TRUE if the data browser confirms insertions; otherwise returns FALSE. See the "confirmInsert" [p. 45] property for details.

bool QDataBrowser::confirmUpdate () const

Returns TRUE if the browser confirms updates; otherwise returns FALSE. See the "confirmUpdate" [p. 45] property for details.

void QDataBrowser::currentChanged (const QSqlRecord * record) [signal]

This signal is emitted whenever the current cursor position changes. The *record* parameter points to the contents of the current cursor's record.

bool QDataBrowser::currentEdited () [virtual protected]

Returns TRUE if the form's edit buffer differs from the current cursor buffer, otherwise FALSE is returned.

void QDataBrowser::cursorChanged (QSqlCursor::Mode mode) [signal]

This signal is emitted whenever the cursor record was changed due to navigation. The *mode* parameter is the edit that just took place, e.g. Insert, Update or Delete. See QSqlCursor::Mode.

void QDataBrowser::del () [virtual slot]

Performs a delete operation on the data browser's cursor. If there is no default cursor or no default form, nothing happens.

Otherwise, the following happens:

The current form's record is deleted from the database, providing that the data browser is not in insert mode. If the data browser is actively inserting a record (see insert()), the insert action is cancelled, and the browser navigates to the last valid record that was current. If there is an error, handleError() is called.

bool QDataBrowser::deleteCurrent () [virtual protected]

Performs a delete on the default cursor using the values from the default form and updates the default form. If there is no default form or no default cursor, nothing happens. If the deletion was successful, the cursor is repositioned to the nearest record and TRUE is returned. The nearest record is the next record if there is one otherwise the previous record if there is one. If an error occurred during the deletion from the database, handleError() is called and FALSE is returned.

See also cursor [Widgets with Qt], form() [p. 39] and handleError() [p. 39].

QString QDataBrowser::filter () const

Returns the data browser's filter. See the "filter" [p. 45] property for details.

void QDataBrowser::first () [virtual slot]

Moves the default cursor to the first record and refreshes the default form to display this record. If there is no default form or no default cursor, nothing happens. If the data browser successfully navigated to the first record, the default cursor is primed for update and the primeUpdate() signal is emitted.

If the browser is already positioned on the first record nothing happens.

void QDataBrowser::firstRecordAvailable (bool available) [signal]

This signal is emitted whenever the position of the cursor changes. The *available* parameter indicates whether or not the first record in the default cursor is available.

QSqlForm * QDataBrowser::form ()

Returns a pointer to the data browser's default form or 0 if no form has been set.

void QDataBrowser::handleError (const QSqlError & error) [virtual protected]

Virtual function which handles the error *error*. The default implementation warns the user with a message box.

void QDataBrowser::insert () [virtual slot]

Performs an insert operation on the data browser's cursor. If there is no default cursor or no default form, nothing happens.

If auto-editing is on (see setAutoEdit()), the following happens:

- If the browser is already actively inserting a record, the current form's data is inserted into the database.
- If the browser is not inserting a record, but the current record was changed by the user, the record is updated in the database with the current form's data (i.e. with the changes).

If there is an error handling any of the above auto-edit actions, handleError() is called and no insert or update is performed.

If no error occurred, or auto-editing is not enabled, the data browser begins actively inserting a record into the database by performing the following actions:

- The default cursor is primed for insert using QSqlCursor::primeInsert().
- The primeInsert() signal is emitted.
- The form is updated with the values in the default cursor's. edit buffer so that the user can fill in the values to be inserted.

bool QDataBrowser::insertCurrent () [virtual protected]

Reads the fields from the default form into the default cursor and performs an insert on the default cursor. If there is no default form or no default cursor, nothing happens. If an error occurred during the insert into the database, handleError() is called and FALSE is returned. If the insert was successfull, the cursor is refreshed and relocated to the newly inserted record, the cursorChanged() signal is emitted, and TRUE is returned.

See also cursorChanged() [p. 38], sqlCursor() [p. 43], form() [p. 39] and handleError() [p. 39].

bool QDataBrowser::isReadOnly () const

Returns TRUE if the browser is read-only; otherwise returns FALSE. See the "readOnly" [p. 45] property for details.

void QDataBrowser::last () [virtual slot]

Moves the default cursor to the last record and refreshes the default form to display this record. If there is no default form or no default cursor, nothing happens. If the data browser successfully navigated to the last record, the default cursor is primed for update and the primeUpdate() signal is emitted.

If the browser is already positioned on the last record nothing happens.

void QDataBrowser::lastRecordAvailable (bool available) [signal]

This signal is emitted whenever the position of the cursor changes. The *available* parameter indicates whether or not the last record in the default cursor is available.

void QDataBrowser::next () [virtual slot]

Moves the default cursor to the next record and refreshes the default form to display this record. If there is no default form or no default cursor, nothing happens. If the data browser successfully navigated to the next record, the default cursor is primed for update and the primeUpdate() signal is emitted.

If the browser is positioned on the last record nothing happens.

void QDataBrowser::nextRecordAvailable (bool available) [signal]

This signal is emitted whenever the position of the cursor changes. The *available* parameter indicates whether or not the next record in the default cursor is available.

void QDataBrowser::prev () [virtual slot]

Moves the default cursor to the previous record and refreshes the default form to display this record. If there is no default form or no default cursor, nothing happens. If the data browser successfully navigated to the previous record, the default cursor is primed for update and the primeUpdate() signal is emitted.

If the browser is positioned on the first record nothing happens.

void QDataBrowser::prevRecordAvailable (bool available) [signal]

This signal is emitted whenever the position of the cursor changes. The *available* parameter indicates whether or not the previous record in the default cursor is available.

void QDataBrowser::primeDelete (QSqlRecord * buf) [signal]

This signal is emitted when the data browser enters deletion mode. The *buf* parameter points to the record buffer being deleted. (Note that QSqlCursor::primeDelete() is *not* called on the default cursor, as this would corrupt values in the form.) Connect to this signal in order to, for example, save a copy of the deleted record for auditing purposes.

See also del() [p. 38].

void QDataBrowser::primeInsert (QSqlRecord * buf) [signal]

This signal is emitted when the data browser enters insertion mode. The *buf* parameter points to the record buffer that is to be inserted. Connect to this signal to, for example, prime the record buffer with default data values, auto-numbered fields etc. (Note that QSqlCursor::primeInsert() is *not* called on the default cursor, as this would corrupt values in the form.)

See also insert() [p. 39].

void QDataBrowser::primeUpdate (QSqlRecord * buf) [signal]

This signal is emitted when the data browser enters update mode. Note that during naviagtion (first(), last(), next(), prev()), each record that is shown in the default form is primed for update. The *buf* parameter points to the record buffer being updated. (Note that QSqlCursor::primeUpdate() is *not* called on the default cursor, as this would corrupt values in the form.) Connect to this signal in order to, for example, keep track of which records have been updated, perhaps for auditing purposes.

See also update() [p. 43].

void QDataBrowser::readFields () [virtual slot]

Reads the fields from the default cursor's edit buffer and displays them in the form. If there is no default cursor or no default form, nothing happens.

void QDataBrowser::refresh () [virtual slot]

Refreshes the data browser's data using the default cursor. The browser's current filter and sort are applied if they have been set.

See also filter [p. 45] and sort [p. 45].

bool QDataBrowser::seek (int i, bool relative = FALSE) [virtual]

Moves the default cursor to the record specified by the index *i* and refreshes the default form to display this record. If there is no default form or no default cursor, nothing happens. If *relative* is TRUE (the default is FALSE), the cursor is moved relative to its current position. If the data browser successfully navigated to the desired record, the default cursor is primed for update and the primeUpdate() signal is emitted.

If the browser is already positioned on the desired record nothing happens.

void QDataBrowser::setAutoEdit (bool autoEdit) [virtual]

Sets whether the browser automatically applies edits to *autoEdit*. See the "autoEdit" [p. 44] property for details.

void QDataBrowser::setBoundaryChecking (bool active)

Sets whether boundary checking is active to *active*. See the "boundaryChecking" [p. 44] property for details.

void QDataBrowser::setConfirmCancels (bool confirm) [virtual]

Sets whether the browser confirms cancel operations to *confirm*. See the "confirmCancels" [p. 44] property for details.

void QDataBrowser::setConfirmDelete (bool confirm) [virtual]

Sets whether the browser confirms deletions to *confirm*. See the "confirmDelete" [p. 44] property for details.

void QDataBrowser::setConfirmEdits (bool confirm) [virtual]

Sets whether the browser confirms edit operations to *confirm*. See the "confirmEdits" [p. 44] property for details.

void QDataBrowser::setConfirmInsert (bool confirm) [virtual]

Sets whether the data browser confirms insertions to *confirm*. See the "confirmInsert" [p. 45] property for details.

void QDataBrowser::setConfirmUpdate (bool confirm) [virtual]

Sets whether the browser confirms updates to *confirm*. See the "confirmUpdate" [p. 45] property for details.

void QDataBrowser::setFilter (const QString & filter)

Sets the data browser's filter to *filter*. See the "filter" [p. 45] property for details.

void QDataBrowser::setForm (QSqlForm * form) [virtual]

Sets the browser's default form to *form*. The cursor and all navigation and data manipulation functions that the browser provides become available to the *form*.

void QDataBrowser::setReadOnly (bool active) [virtual]

Sets whether the browser is read-only to *active*. See the "readOnly" [p. 45] property for details.

void QDataBrowser::setSort (const QStringList & sort)

Sets the data browser's sort to *sort*. See the "sort" [p. 45] property for details.

void QDataBrowser::setSort (const QSqlIndex & sort)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the data browser's sort to the QSqlIndex *sort*. To apply the new sort, use refresh().

void QDataBrowser::setSqlCursor (QSqlCursor * cursor, bool autoDelete = FALSE) [virtual]

Sets the default cursor used by the data browser to *cursor*. If *autoDelete* is TRUE (the default is FALSE), the data browser takes ownership of the *cursor* pointer, which will be deleted when the browser is destroyed, or when setSqlCursor() is called again. To activate the *cursor* use refresh(). The cursor's edit buffer is used in the default form to browse and edit records.

See also sqlCursor() [p. 43], form() [p. 39] and setForm() [p. 42].

QStringList QDataBrowser::sort () const

Returns the data browser's sort. See the "sort" [p. 45] property for details.

QSqlCursor * QDataBrowser::sqlCursor () const

Returns a pointer to the default cursor used for navigation, or 0 if there is no default cursor.

See also setSqlCursor() [p. 43].

void QDataBrowser::update () [virtual slot]

Performs an update operation on the data browser's cursor.

If there is no default cursor or no default form, nothing happens. Otherwise, the following happens:

If the data browser is actively inserting a record (see insert()), that record is inserted into the database using insertCurrent(). Otherwise, the database is updated with the current form's data using updateCurrent(). If there is an error handling either action, handleError() is called.

void QDataBrowser::updateBoundary () [slot]

If boundaryChecking() is TRUE, checks the boundary of the current default cursor and emits signals which indicate the position of the cursor.

bool QDataBrowser::updateCurrent () [virtual protected]

Reads the fields from the default form into the default cursor and performs an update on the default cursor. If there is no default form or no default cursor, nothing happens. If an error occurred during the update on the database, handleError() is called and FALSE is returned. If the update was successfull, the cursor is refreshed and relocated to the updated record, the cursorChanged() signal is emitted, and TRUE is returned.

See also cursor [Widgets with Qt], form() [p. 39] and handleError() [p. 39].

void QDataBrowser::writeFields () [virtual slot]

Writes the form's data to the default cursor's edit buffer. If there is no default cursor or no default form, nothing happens.

Property Documentation

bool autoEdit

This property holds whether the browser automatically applies edits.

The default value for this property is TRUE. When the user begins an insertion or an update on a form there are two possible outcomes when they navigate to another record:

- the insert or update is is performed — this occurs if autoEdit is TRUE
- the insert or update is discarded — this occurs if autoEdit is FALSE

Set this property's value with setAutoEdit() and get this property's value with autoEdit().

bool boundaryChecking

This property holds whether boundary checking is active.

When boundary checking is active (the default), signals are emitted indicating the current position of the default cursor.

See also boundary() [p. 37].

Set this property's value with setBoundaryChecking() and get this property's value with boundaryChecking().

bool confirmCancels

This property holds whether the browser confirms cancel operations.

If this property is TRUE, all cancels must be confirmed by the user through a message box (this behavior can be changed by overriding the confirmCancel() function), otherwise all cancels occur immediately. The default is FALSE.

See also confirmEdits [p. 44] and confirmCancel() [p. 37].

Set this property's value with setConfirmCancels() and get this property's value with confirmCancels().

bool confirmDelete

This property holds whether the browser confirms deletions.

If this property is TRUE, the browser confirms deletions, otherwise deletions happen immediately.

See also confirmCancels [p. 44], confirmEdits [p. 44], confirmUpdate [p. 45], confirmInsert [p. 45] and confirmEdit() [p. 37].

Set this property's value with setConfirmDelete() and get this property's value with confirmDelete().

bool confirmEdits

This property holds whether the browser confirms edit operations.

If this property is TRUE, the browser confirms all edit operations (insertions, updates and deletions), otherwise all edit operations happen immediately. Confirmation is achieved by presenting the user with a message box — this behavior can be changed by reimplementing the confirmEdit() function,

See also confirmEdit() [p. 37], confirmCancels [p. 44], confirmInsert [p. 45], confirmUpdate [p. 45] and confirmDelete [p. 44].

Set this property's value with setConfirmEdits() and get this property's value with confirmEdits().

bool confirmInsert

This property holds whether the data browser confirms insertions.

If this property is TRUE, the browser confirms insertions, otherwise insertions happen immediately.

See also confirmCancels [p. 44], confirmEdits [p. 44], confirmUpdate [p. 45], confirmDelete [p. 44] and confirmEdit() [p. 37].

Set this property's value with setConfirmInsert() and get this property's value with confirmInsert().

bool confirmUpdate

This property holds whether the browser confirms updates.

If this property is TRUE, the browser confirms updates, otherwise updates happen immediately.

See also confirmCancels [p. 44], confirmEdits [p. 44], confirmInsert [p. 45], confirmDelete [p. 44] and confirmEdit() [p. 37].

Set this property's value with setConfirmUpdate() and get this property's value with confirmUpdate().

QString filter

This property holds the data browser's filter.

The filter applies to the data shown in the browser. Call refresh() to apply the new filter. A filter is a string containing a SQL WHERE clause without the WHERE keyword, e.g. "id>1000", "name LIKE 'A%'".

There is no default filter.

See also sort [p. 45].

Set this property's value with setFilter() and get this property's value with filter().

bool readOnly

This property holds whether the browser is read-only.

The default is FALSE, i.e. data can be edited. If the data browser is read-only, no database edits will be allowed.

Set this property's value with setReadOnly() and get this property's value with isReadOnly().

QStringList sort

This property holds the data browser's sort.

The data browser's sort affects the order in which records are viewed in the browser. Call refresh() to apply the new sort.

When retrieving the sort property, a string list is returned in the form 'fieldname order', e.g. 'id ASC', 'surname DESC'.

There is no default sort.

See also filter [p. 45] and refresh() [p. 41].

Set this property's value with setSort() and get this property's value with sort().

QDataTable Class Reference

The QDataTable class provides a flexible SQL table widget that supports browsing and editing.

This class is part of the **sql** module.

`#include <qdatatable.h>`

Inherits QTable [Widgets with Qt].

Public Members

- **QDataTable** (QWidget * parent = 0, const char * name = 0)
- **QDataTable** (QSqlCursor * cursor, bool autoPopulate = FALSE, QWidget * parent = 0, const char * name = 0)
- **~QDataTable** ()
- virtual void **addColumn** (const QString & fieldName, const QString & label = QString::null, int width = -1, const QIconSet & iconset = QIconSet ( ))
- virtual void **removeColumn** (uint col)
- virtual void **setColumn** (uint col, const QString & fieldName, const QString & label = QString::null, int width = -1, const QIconSet & iconset = QIconSet ( ))
- QString **nullText** () const
- QString **trueText** () const
- QString **falseText** () const
- DateFormat **dateFormat** () const
- bool **confirmEdits** () const
- bool **confirmInsert** () const
- bool **confirmUpdate** () const
- bool **confirmDelete** () const
- bool **confirmCancels** () const
- bool **autoDelete** () const
- bool **autoEdit** () const
- QString **filter** () const
- QStringList **sort** () const
- virtual void **setSqlCursor** (QSqlCursor * cursor = 0, bool autoPopulate = FALSE, bool autoDelete = FALSE)
- QSqlCursor * **sqlCursor** () const
- virtual void **setNullText** (const QString & nullText)
- virtual void **setTrueText** (const QString & trueText)
- virtual void **setFalseText** (const QString & falseText)
- virtual void **setDateFormat** (const DateFormat f)
- virtual void **setConfirmEdits** (bool confirm)
- virtual void **setConfirmInsert** (bool confirm)
- virtual void **setConfirmUpdate** (bool confirm)

- virtual void **setConfirmDelete** (bool confirm)
- virtual void **setConfirmCancels** (bool confirm)
- virtual void **setAutoDelete** (bool enable)
- virtual void **setAutoEdit** (bool autoEdit)
- virtual void **setFilter** (const QString & filter)
- virtual void **setSort** (const QStringList & sort)
- virtual void **setSort** (const QSqlIndex & sort)
- enum **Refresh** { RefreshData = 1, RefreshColumns = 2, RefreshAll = 3 }
- void **refresh** (Refresh mode)
- virtual void **sortColumn** (int col, bool ascending = TRUE, bool wholeRows = FALSE)
- virtual QString **text** (int row, int col) const
- QVariant **value** (int row, int col) const
- QSqlRecord * **currentRecord** () const
- void **installEditorFactory** (QSqlEditorFactory * f)
- void **installPropertyMap** (QSqlPropertyMap * m)
- virtual int **numCols** () const
- virtual int **numRows** () const

Public Slots

- virtual void **find** (const QString & str, bool caseSensitive, bool backwards)
- virtual void **sortAscending** (int col)
- virtual void **sortDescending** (int col)
- virtual void **refresh** ()

Signals

- void **currentChanged** (QSqlRecord * record)
- void **primeInsert** (QSqlRecord * buf)
- void **primeUpdate** (QSqlRecord * buf)
- void **primeDelete** (QSqlRecord * buf)
- void **beforeInsert** (QSqlRecord * buf)
- void **beforeUpdate** (QSqlRecord * buf)
- void **beforeDelete** (QSqlRecord * buf)
- void **cursorChanged** (QSql::Op mode)

Properties

- bool **autoEdit** — whether the data table automatically applies edits
- bool **confirmCancels** — whether the data table confirms cancel operations
- bool **confirmDelete** — whether the data table confirms delete operations
- bool **confirmEdits** — whether the data table confirms edit operations
- bool **confirmInsert** — whether the data table confirms insert operations
- bool **confirmUpdate** — whether the data table confirms update operations
- DateFormat **dateFormat** — the format how date/time values are displayed
- QString **falseText** — the text used to represent false values
- QString **filter** — the data filter for the data table

- QString **nullText** — the text used to represent NULL values
- int **numCols** — the number of columns in the table *(read only)*
- int **numRows** — the number of rows in the table *(read only)*
- QStringList **sort** — the data table's sort
- QString **trueText** — the text used to represent true values

Protected Members

- virtual bool **insertCurrent** ()
- virtual bool **updateCurrent** ()
- virtual bool **deleteCurrent** ()
- virtual QSql::Confirm **confirmEdit** (QSql::Op m)
- virtual QSql::Confirm **confirmCancel** (QSql::Op m)
- virtual void **handleError** (const QSqlError & e)
- virtual bool **beginInsert** ()
- virtual QWidget * **beginUpdate** (int row, int col, bool replace)
- int **indexOf** (uint i) const
- void **reset** ()
- void **setSize** (QSqlCursor * sql)
- virtual void **paintField** (QPainter * p, const QSqlField * field, const QRect & cr, bool selected)
- virtual int **fieldAlignment** (const QSqlField * field)

Detailed Description

The QDataTable class provides a flexible SQL table widget that supports browsing and editing.

QDataTable supports various functions for presenting and editing SQL data from a QSqlCursor in a table.

If you want a to present your data in a form use QDataBrowser, or for read-only forms, QDataView.

When displaying data, QDataTable only retrieves data for visible rows. If the driver supports the 'query size' property the QDataTable will have the correct number of rows and the vertical scrollbar will accurately reflect the number of rows displayed in proportion to the number of rows in the dataset. If the driver does not support the 'query size' property rows are dynamically fetched from the database on an as-needed basis with the scrollbar becoming more accurate as the user scrolls down through the records. This allows extremely large queries to be displayed as quickly as possible, with minimum memory usage.

QDataTable inherits QTable's API and extends it with functions to sort and filter the data and sort columns. See setSqlCursor(), setFilter(), setSort(), setSorting(), sortColumn() and refresh().

When displaying editable cursors, cell editing will be enabled. (For more information on editable cursors, see QSqlCursor). QDataTable can be used to modify existing data and to add new records. When a user makes changes to a field in the table, the cursor's edit buffer is used. The table will not send changes in the edit buffer to the database until the user moves to a different record in the grid or presses Return. Cell editing is initiated by pressing F2 (or right clicking and then clicking the appropriate popup menu item) and cancelled by pressing Esc. If there is a problem updating or adding data, errors are handled automatically (see handleError() to change this behavior). Note that if autoEdit() is FALSE navigating to another record will cancel the insert or update.

The user can be asked to confirm all edits with setConfirmEdits(). For more precise control use setConfirmInsert(), setConfirmUpdate(), setConfirmDelete() and setConfirmCancels(). Use setAutoEdit() to control the behaviour of the table when the user edits a record and then navigates. (Note that setAutoDelete() is unrelated; it is used to set whether the QSqlCursor is deleted when the table is deleted.)

Since the data table can perform edits, it must be able to uniquely identify every record so that edits are correctly applied. Because of this the underlying cursor must have a valid primary index to ensure that a unique record is inserted, updated or deleted within the database otherwise the database may be changed to an inconsistent state.

QDataTable creates editors using the default QSqlEditorFactory. Different editor factories can be used by calling installEditorFactory(). A property map is used to map between the cell's value and the editor. You can use your own property map with installPropertyMap().

The contents of a cell is available as a QString with text() or as a QVariant with value(). The current record is returned by currentRecord(). Use the find() function to search for a string in the table.

Editing actions can be applied programatically. For example, the insertCurrent() function reads the fields from the current record into the cursor and performs the insert. The updateCurrent() and deleteCurrent() functions perform similarly to update and delete the current record respectively.

Columns in the table can be created automatically based on the cursor (see setSqlCursor()). Columns can be manipulated manually using addColumn(), removeColumn() and setColumn().

The table automatically copies many of the properties of the cursor to format the display of data within cells (alignment, visibility, etc.). The cursor can be changed with setSqlCursor(). The filter (see setFilter()) and sort defined within the table are used instead of the filter and sort set on the cursor. For sorting options see setSort(), sortColumn(), sortAscending() and sortDescending().

The text used to represent NULL, TRUE and FALSE values can be changed with setNullText(), setTrueText() and setFalseText() respectively. You can change the appearance of cells by reimplementing paintField().

Whenever a new row is selected in the table the currentChanged() signal is emitted. The primeInsert() signal is emitted when an insert is initiated. The primeUpdate() and primeDelete() signals are emitted when update and deletion are initiated respectively. Just before the database is updated a signal is emitted; beforeInsert(), beforeUpdate() or beforeDelete() as appropriate.

See also Database Classes.

Member Type Documentation

QDataTable::Refresh

This enum describes the refresh options.

The currently defined values are:

- `QDataTable::RefreshData` - refresh the data, i.e. read it from the database
- `QDataTable::RefreshColumns` - refresh the list of fields, e.g. the column headings
- `QDataTable::RefreshAll` - refresh both the data and the list of fields

Member Function Documentation

QDataTable::QDataTable (QWidget * parent = 0, const char * name = 0)

Constructs a data table which is a child of *parent*, with the name *name*.

QDataTable::QDataTable (QSqlCursor * cursor, bool autoPopulate = FALSE, QWidget * parent = 0, const char * name = 0)

Constructs a data table which is a child of *parent*, with the name *name* using the cursor *cursor*.

If *autoPopulate* is TRUE (the default is FALSE), columns are automatically created based upon the fields in the *cursor* record. Note that *autoPopulate* only governs the creation of columns; to load the cursor's data into the table use refresh().

If the *cursor* is read-only, the table also becomes read-only. In addition, the table adopts the cursor's driver's definition for representing NULL values as strings.

QDataTable::~QDataTable ()

Destroys the object and frees any allocated resources.

void QDataTable::addColumn (const QString & fieldName, const QString & label = QString::null, int width = -1, const QIconSet & iconset = QIconSet ( )) [virtual]

Adds the next column to be displayed using the field *fieldName*, column label *label*, width *width* and iconset *iconset*.

If *label* is specified, it is used as the column's header label, otherwise the field's display label is used when setSqlCursor() is called. The *iconset* is used to set the icon used by the column header; by default there is no icon.

See also setSqlCursor() [p. 58] and refresh() [p. 55].

Examples: sql/overview/subclass1/main.cpp, sql/overview/subclass3/main.cpp, sql/overview/table2/main.cpp and sql/sqltable/main.cpp.

bool QDataTable::autoDelete () const

Returns TRUE if the table will automatically delete the cursor specified by setSqlCursor(), otherwise returns FALSE.

bool QDataTable::autoEdit () const

Returns TRUE if the data table automatically applies edits; otherwise returns FALSE. See the "autoEdit" [p. 59] property for details.

void QDataTable::beforeDelete (QSqlRecord * buf) [signal]

This signal is emitted just before the currently selected record is deleted from the database. The *buf* parameter points to the edit buffer being deleted. Connect to this signal to, for example, copy some of the fields for later use.

void QDataTable::beforeInsert (QSqlRecord * buf) [signal]

This signal is emitted just before the cursor's edit buffer is inserted into the database. The *buf* parameter points to the edit buffer being inserted. Connect to this signal to, for example, populate a key field with a unique sequence number.

void QDataTable::beforeUpdate (QSqlRecord * buf) [signal]

This signal is emitted just before the cursor's edit buffer is updated in the database. The *buf* parameter points to the edit buffer being updated. Connect to this signal when you want to transform the user's data behind-the-scenes.

bool QDataTable::beginInsert () [virtual protected]

Protected virtual function called when editing is about to begin on a new record. If the table is read-only, or if there's no cursor or the cursor does not allow inserts, nothing happens.

Editing takes place using the cursor's edit buffer(see QSqlCursor::editBuffer()).

When editing begins, a new row is created in the table marked with an asterisk '*' in the row's vertical header column, i.e. at the left of the row.

QWidget * QDataTable::beginUpdate (int row, int col, bool replace) [virtual protected]

Protected virtual function called when editing is about to begin on an existing row. If the table is read-only, or if there's no cursor, nothing happens.

Editing takes place using the cursor's edit buffer (see QSqlCursor::editBuffer()).

row and *col* refer to the row and column in the QDataTable.

(*replace* is provided for reimplementors and reflects the API of QTable::beginEdit().)

QSql::Confirm QDataTable::confirmCancel (QSql::Op m) [virtual protected]

Protected virtual function which returns a confirmation for cancelling an edit mode of *m*. Derived classes can reimplement this function to provide their own cancel dialog. The default implementation uses a message box which prompts the user to confirm the cancel.

bool QDataTable::confirmCancels () const

Returns TRUE if the data table confirms cancel operations; otherwise returns FALSE. See the "confirmCancels" [p. 59] property for details.

bool QDataTable::confirmDelete () const

Returns TRUE if the data table confirms delete operations; otherwise returns FALSE. See the "confirmDelete" [p. 59] property for details.

QSql::Confirm QDataTable::confirmEdit (QSql::Op m) [virtual protected]

Protected virtual function which returns a confirmation for an edit of mode *m*. Derived classes can reimplement this function to provide their own confirmation dialog. The default implementation uses a message box which prompts the user to confirm the edit action.

bool QDataTable::confirmEdits () const

Returns TRUE if the data table confirms edit operations; otherwise returns FALSE. See the "confirmEdits" [p. 60] property for details.

bool QDataTable::confirmInsert () const

Returns TRUE if the data table confirms insert operations; otherwise returns FALSE. See the "confirmInsert" [p. 60] property for details.

bool QDataTable::confirmUpdate () const

Returns TRUE if the data table confirms update operations; otherwise returns FALSE. See the "confirmUpdate" [p. 60] property for details.

void QDataTable::currentChanged (QSqlRecord * record) [signal]

This signal is emitted whenever a new row is selected in the table. The *record* parameter points to the contents of the newly selected record.

QSqlRecord * QDataTable::currentRecord () const

Returns a pointer to the currently selected record, or 0 if there is no current selection. The table owns the pointer, so do *not* delete it or otherwise modify it or the cursor it points to.

void QDataTable::cursorChanged (QSql::Op mode) [signal]

This signal is emitted whenever the cursor record was changed due to an edit. The *mode* parameter is the type of edit that just took place.

DateFormat QDataTable::dateFormat () const

Returns the format how date/time values are displayed. See the "dateFormat" [p. 60] property for details.

bool QDataTable::deleteCurrent () [virtual protected]

For an editable table, issues a delete on the current cursor's primary index using the values of the currently selected row. If there is no current cursor or there is no current selection, nothing happens. If confirmEdits() or confirmDelete() is TRUE, confirmEdit() is called to confirm the delete. Returns TRUE if the delete succeeded, otherwise FALSE.

The underlying cursor must have a valid primary index to ensure that a unique record is deleted within the database otherwise the database may be changed to an inconsistent state.

QString QDataTable::falseText () const

Returns the text used to represent false values. See the "falseText" [p. 60] property for details.

int QDataTable::fieldAlignment (const QSqlField * field) [virtual protected]

Returns the alignment for *field*.

QString QDataTable::filter () const

Returns the data filter for the data table. See the "filter" [p. 60] property for details.

void QDataTable::find (const QString & str, bool caseSensitive, bool backwards) [virtual slot]

Searches the current cursor for a cell containing the string *str* starting at the current cell and working forwards (or backwards if *backwards* is TRUE). If the string is found, the cell containing the string is set as the current cell. If *caseSensitive* is FALSE the case of *str* will be ignored.

The search will wrap, i.e. if the first (or if backwards is TRUE, last) cell is reached without finding *str* the search will continue until it reaches the starting cell. If *str* is not found the search will fail and the current cell will remain unchanged.

void QDataTable::handleError (const QSqlError & e) [virtual protected]

Protected virtual function which is called when an error *e* has occurred on the current cursor(). The default implementation displays a warning message to the user with information about the error.

int QDataTable::indexOf (uint i) const [protected]

Returns the index of the field within the current SQL query that is displayed in column *i*.

bool QDataTable::insertCurrent () [virtual protected]

For an editable table, issues an insert on the current cursor using the values in the cursor's edit buffer. If there is no current cursor or there is no current "insert" row, nothing happens. If confirmEdits() or confirmInsert() is TRUE, confirmEdit() is called to confirm the insert. Returns TRUE if the insert succeeded, otherwise returns FALSE.

The underlying cursor must have a valid primary index to ensure that a unique record is inserted within the database otherwise the database may be changed to an inconsistent state.

void QDataTable::installEditorFactory (QSqlEditorFactory * f)

Installs a new SQL editor factory *f*. This enables the user to create and instantiate their own editors for use in cell editing. Note that QDataTable takes ownership of this pointer, and will delete it when it is no longer needed or when installEditorFactory() is called again.

See also QSqlEditorFactory [p. 92].

void QDataTable::installPropertyMap (QSqlPropertyMap * m)

Installs a new property map *m*. This enables the user to create and instantiate their own property maps for use in cell editing. Note that QDataTable takes ownership of this pointer, and will delete it when it is no longer needed or when installPropertMap() is called again.

See also QSqlPropertyMap [p. 112].

QString QDataTable::nullText () const

Returns the text used to represent NULL values. See the "nullText" [p. 61] property for details.

int QDataTable::numCols () const [virtual]

Returns the number of columns in the table. See the "numCols" [p. 61] property for details.

Reimplemented from QTable [Widgets with Qt].

int QDataTable::numRows () const [virtual]

Returns the number of rows in the table. See the "numRows" [p. 61] property for details.

Reimplemented from QTable [Widgets with Qt].

void QDataTable::paintField (QPainter * p, const QSqlField * field, const QRect & cr, bool selected) [virtual protected]

Paints the *field* on the painter *p*. The painter has already been translated to the appropriate cell's origin where the *field* is to be rendered. *cr* describes the cell coordinates in the content coordinate system. The *selected* parameter is ignored.

If you want to draw custom field content you have to reimplement paintField() to do the custom drawing. The default implementation renders the *field* value as text. If the field is NULL, nullText() is displayed in the cell. If the field is Boolean, trueText() or falseText() is displayed as appropriate.

Example: sql/overview/table4/main.cpp.

void QDataTable::primeDelete (QSqlRecord * buf) [signal]

This signal is emitted after the cursor is primed for delete by the table, when a delete action is beginning on the table. The *buf* parameter points to the edit buffer being deleted. Connect to this signal in order to, for example, record auditing information on deletions.

void QDataTable::primeInsert (QSqlRecord * buf) [signal]

This signal is emitted after the cursor is primed for insert by the table, when an insert action is beginning on the table. The *buf* parameter points to the edit buffer being inserted. Connect to this signal in order to, for example, prime the record buffer with default data values.

void QDataTable::primeUpdate (QSqlRecord * buf) [signal]

This signal is emitted after the cursor is primed for update by the table, when an update action is beginning on the table. The *buf* parameter points to the edit buffer being updated. Connect to this signal in order to, for example, provide some visual feedback that the user is in 'insert mode'.

void QDataTable::refresh () [virtual slot]

Refreshes the table. The cursor is refreshed using the current filter, the current sort, and the currently defined columns. Equivalent to calling refresh(QDataTable::RefreshData).

Examples: sql/overview/subclass1/main.cpp, sql/overview/table1/main.cpp, sql/overview/table2/main.cpp and sql/sqltable/main.cpp.

void QDataTable::refresh (Refresh mode)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Refreshes the table. If there is no currently defined cursor (see setSqlCursor()), nothing happens. The *mode* parameter determines which type of refresh will take place.

See also Refresh [p. 50], setSqlCursor() [p. 58] and addColumn() [p. 51].

void QDataTable::removeColumn (uint col) [virtual]

Removes column *col* from the list of columns to be displayed. If *col* does not exist, nothing happens.

See also QSqlField [p. 97].

void QDataTable::reset () [protected]

Resets the table so that it displays no data.

See also setSqlCursor() [p. 58].

void QDataTable::setAutoDelete (bool enable) [virtual]

Sets the cursor auto-delete flag to *enable*. If *enable* is TRUE, the table will automatically delete the cursor specified by setSqlCursor(). Otherwise, (the default), the cursor will not be deleted.

void QDataTable::setAutoEdit (bool autoEdit) [virtual]

Sets whether the data table automatically applies edits to *autoEdit*. See the "autoEdit" [p. 59] property for details.

void QDataTable::setColumn (uint col, const QString & fieldName, const QString & label = QString::null, int width = -1, const QIconSet & iconset = QIconSet ( )) [virtual]

Sets the *col* column to display using the field *fieldName*, column label *label*, width *width* and iconset *iconset*.

If *label* is specified, it is used as the column's header label, otherwise the field's display label is used when setSqlCursor() is called. The *iconset* is used to set the icon used by the column header; by default there is no icon.

See also setSqlCursor() [p. 58] and refresh() [p. 55].

void QDataTable::setConfirmCancels (bool confirm) [virtual]

Sets whether the data table confirms cancel operations to *confirm*. See the "confirmCancels" [p. 59] property for details.

void QDataTable::setConfirmDelete (bool confirm) [virtual]

Sets whether the data table confirms delete operations to *confirm*. See the "confirmDelete" [p. 59] property for details.

void QDataTable::setConfirmEdits (bool confirm) [virtual]

Sets whether the data table confirms edit operations to *confirm*. See the "confirmEdits" [p. 60] property for details.

void QDataTable::setConfirmInsert (bool confirm) [virtual]

Sets whether the data table confirms insert operations to *confirm*. See the "confirmInsert" [p. 60] property for details.

void QDataTable::setConfirmUpdate (bool confirm) [virtual]

Sets whether the data table confirms update operations to *confirm*. See the "confirmUpdate" [p. 60] property for details.

void QDataTable::setDateFormat (const DateFormat f) [virtual]

Sets the format how date/time values are displayed to *f*. See the "dateFormat" [p. 60] property for details.

void QDataTable::setFalseText (const QString & falseText) [virtual]

Sets the text used to represent false values to *falseText*. See the "falseText" [p. 60] property for details.

void QDataTable::setFilter (const QString & filter) [virtual]

Sets the data filter for the data table to *filter*. See the "filter" [p. 60] property for details.

void QDataTable::setNullText (const QString & nullText) [virtual]

Sets the text used to represent NULL values to *nullText*. See the "nullText" [p. 61] property for details.

void QDataTable::setSize (QSqlCursor * sql) [protected]

If the cursor's *sql* driver supports query sizes, the number of rows in the table is set to the size of the query. Otherwise, the table dynamically resizes itself as it is scrolled. If \q sql is not active, it is made active by issuing a select() on the cursor using the *sql* cursor's current filter and current sort.

void QDataTable::setSort (const QStringList & sort) [virtual]

Sets the data table's sort to *sort*. See the "sort" [p. 61] property for details.

void QDataTable::setSort (const QSqlIndex & sort) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the sort to be applied to the displayed data to *sort*. If there is no current cursor, nothing happens. A QSqlIndex contains field names and their ordering (ASC or DESC); these are used to compose the ORDER BY clause.

See also sort [p. 61].

void QDataTable::setSqlCursor (QSqlCursor * cursor = 0, bool autoPopulate = FALSE, bool autoDelete = FALSE) [virtual]

Sets *cursor* as the data source for the table. To force the display of the data from *cursor*, use refresh(). If *autoPopulate* is TRUE, columns are automatically created based upon the fields in the *cursor* record. If *autoDelete* is TRUE (the default is FALSE), the table will take ownership of the *cursor* and delete it when appropriate. If the *cursor* is read-only, the table becomes read-only. The table adopts the cursor's driver's definition for representing NULL values as strings.

See also refresh() [p. 55], readOnly [Widgets with Qt], setAutoDelete() [p. 56] and QSqlDriver::nullText() [p. 90].

void QDataTable::setTrueText (const QString & trueText) [virtual]

Sets the text used to represent true values to *trueText*. See the "trueText" [p. 61] property for details.

QStringList QDataTable::sort () const

Returns the data table's sort. See the "sort" [p. 61] property for details.

void QDataTable::sortAscending (int col) [virtual slot]

Sorts column *col* in ascending order.

See also sorting [Widgets with Qt].

void QDataTable::sortColumn (int col, bool ascending = TRUE, bool wholeRows = FALSE) [virtual]

Sorts column *col* in ascending order if *ascending* is TRUE (the default), otherwise sorts in descending order. The *wholeRows* parameter is ignored for SQL tables.

Reimplemented from QTable [Widgets with Qt].

void QDataTable::sortDescending (int col) [virtual slot]

Sorts column *col* in descending order.

See also sorting [Widgets with Qt].

QSqlCursor * QDataTable::sqlCursor () const

Returns a pointer to the cursor used by the data table.

QString QDataTable::text (int row, int col) const [virtual]

Returns the text in cell *row*, *col*, or an empty string if the cell is empty. If the cell's value is NULL then nullText() will be returned. If the cell does not exist then a null QString is returned.

Reimplemented from QTable [Widgets with Qt].

QString QDataTable::trueText () const

Returns the text used to represent true values. See the "trueText" [p. 61] property for details.

bool QDataTable::updateCurrent () [virtual protected]

For an editable table, issues an update using the cursor's edit buffer. If there is no current cursor or there is no current selection, nothing happens. If confirmEdits() or confirmUpdate() is TRUE, confirmEdit() is called to confirm the update. Returns TRUE if the update succeeded, otherwise returns FALSE.

The underlying cursor must have a valid primary index to ensure that a unique record is updated within the database otherwise the database may be changed to an inconsistent state.

QVariant QDataTable::value (int row, int col) const

Returns the value in cell *row*, *col*, or an invalid value if the cell does not exist or has no value.

Property Documentation

bool autoEdit

This property holds whether the data table automatically applies edits.

The default value for this property is TRUE. When the user begins an insert or update in the table there are two possible outcomes when they navigate to another record:

1. the insert or update is is performed — this occurs if autoEdit is TRUE
2. the insert or update is abandoned — this occurs if autoEdit is FALSE

Set this property's value with setAutoEdit() and get this property's value with autoEdit().

bool confirmCancels

This property holds whether the data table confirms cancel operations.

If the confirmCancel property is active, all cancels must be confirmed by the user through a message box (this behavior can be changed by overriding the confirmCancel() function), otherwise all cancels occur immediately. The default is FALSE.

See also confirmEdits [p. 60] and confirmCancel() [p. 52].

Set this property's value with setConfirmCancels() and get this property's value with confirmCancels().

bool confirmDelete

This property holds whether the data table confirms delete operations.

If the confirmDelete property is active, all deletions must be confirmed by the user through a message box (this behaviour can be changed by overriding the confirmEdit() function), otherwise all delete operations occur immediately.

See also confirmCancels [p. 59], confirmEdits [p. 60], confirmUpdate [p. 60] and confirmInsert [p. 60].

Set this property's value with setConfirmDelete() and get this property's value with confirmDelete().

bool confirmEdits

This property holds whether the data table confirms edit operations.

If the confirmEdits property is active, the data table confirms all edit operations (inserts, updates and deletes), otherwise all edit operations occur immediately.

See also confirmCancels [p. 59], confirmInsert [p. 60], confirmUpdate [p. 60] and confirmDelete [p. 59].

Set this property's value with setConfirmEdits() and get this property's value with confirmEdits().

bool confirmInsert

This property holds whether the data table confirms insert operations.

If the confirmInsert property is active, all insertions must be confirmed by the user through a message box (this behaviour can be changed by overriding the confirmEdit() function), otherwise all insert operations occur immediately.

See also confirmCancels [p. 59], confirmEdits [p. 60], confirmUpdate [p. 60] and confirmDelete [p. 59].

Set this property's value with setConfirmInsert() and get this property's value with confirmInsert().

bool confirmUpdate

This property holds whether the data table confirms update operations.

If the confirmUpdate property is active, all updates must be confirmed by the user through a message box (this behaviour can be changed by overriding the confirmEdit() function), otherwise all update operations occur immediately.

See also confirmCancels [p. 59], confirmEdits [p. 60], confirmInsert [p. 60] and confirmDelete [p. 59].

Set this property's value with setConfirmUpdate() and get this property's value with confirmUpdate().

DateFormat dateFormat

This property holds the format how date/time values are displayed.

The dateFormat property will be used to display date/time values in the table. The default value is 'Qt::LocalDate'.

Set this property's value with setDateFormat() and get this property's value with dateFormat().

QString falseText

This property holds the text used to represent false values.

The falseText property will be used to represent NULL values in the table. The default value is 'False'.

Set this property's value with setFalseText() and get this property's value with falseText().

QString filter

This property holds the data filter for the data table.

The filter applies to the data shown in the table. To actually a new filter, use refresh(). A filter string is an SQL WHERE clause without the WHERE keyword.

There is no default filter.

See also sort [p. 61].

Set this property's value with setFilter() and get this property's value with filter().

QString nullText

This property holds the text used to represent NULL values.

The nullText property will be used to represent NULL values in the table. The default value is provided by the cursor's driver.

Set this property's value with setNullText() and get this property's value with nullText().

int numCols

This property holds the number of columns in the table.

Get this property's value with numCols().

int numRows

This property holds the number of rows in the table.

Get this property's value with numRows().

QStringList sort

This property holds the data table's sort.

The table's sort affects the order in which data records are displayed in the table. To apply a sort, use refresh().

When examining the sort property, a string list is returned with each item having the form 'fieldname order' (e.g., 'id ASC', 'surname DESC').

There is no default sort.

See also filter [p. 60] and refresh() [p. 55].

Set this property's value with setSort() and get this property's value with sort().

QString trueText

This property holds the text used to represent true values.

The trueText property will be used to represent NULL values in the table. The default value is 'True'.

Set this property's value with setTrueText() and get this property's value with trueText().

QDataView Class Reference

The QDataView class provides read-only SQL forms.

This class is part of the **sql** module.

`#include <qdataview.h>`

Inherits QWidget [Widgets with Qt].

Public Members

- **QDataView** (QWidget * parent = 0, const char * name = 0, WFlags fl = 0)
- **~QDataView** ()
- virtual void **setForm** (QSqlForm * form)
- QSqlForm * **form** ()
- virtual void **setRecord** (QSqlRecord * record)
- QSqlRecord * **record** ()

Public Slots

- virtual void **refresh** (QSqlRecord * buf)
- virtual void **readFields** ()
- virtual void **writeFields** ()
- virtual void **clearValues** ()

Detailed Description

The QDataView class provides read-only SQL forms.

This class provides a form which displays SQL field data from a record buffer. Because QDataView does not support editing it uses less resources than a QDataBrowser. This class is well suited for displaying read-only data from a SQL database.

If you want a to present your data in an editable form use QDataBrowser; if you want a table-based presentation of your data use QDataTable.

The form is associated with the data view with setForm() and the record is associated with setRecord(). You can also pass a QSqlRecord to the refresh() function which will set the record to the given record and read the record's fields into the form.

See also Database Classes.

Member Function Documentation

QDataView::QDataView (QWidget * parent = 0, const char * name = 0, WFlags fl = 0)

Constructs a data view which is a child of *parent*, with the name *name* and widget flags set to *fl*.

QDataView::~QDataView ()

Destroys the object and frees any allocated resources.

void QDataView::clearValues () [virtual slot]

Clears the default form's values. If there is no default form, nothing happens. All the values are set to their 'zero state', e.g. 0 for numeric fields, "" for string fields.

QSqlForm * QDataView::form ()

Returns the default form used by the data view, or 0 if there is none.

See also setForm() [p. 63].

void QDataView::readFields () [virtual slot]

Causes the default form to read its fields from the record buffer. If there is no default form, or no record, nothing happens.

See also setForm() [p. 63].

QSqlRecord * QDataView::record ()

Returns the default record used by the data view, or 0 if there is none.

See also setRecord() [p. 64].

void QDataView::refresh (QSqlRecord * buf) [virtual slot]

Causes the default form to display the contents of *buf*. If there is no default form, nothing happens.The *buf* also becomes the default record for all subsequent calls to readFields() and writefields(). This slot is equivalant to calling:

```
myView.setRecord( record );
myView.readFields();
```

See also setRecord() [p. 64] and readFields() [p. 63].

void QDataView::setForm (QSqlForm * form) [virtual]

Sets the form used by the data view to *form*. If a record has already been assigned to the data view, the form will display that record's data.

See also form() [p. 63].

void QDataView::setRecord (QSqlRecord * record) [virtual]

Sets the record used by the data view to *record*. If a form has already been assigned to the data view, the form will display the data from *record* in that form.

See also record() [p. 63].

void QDataView::writeFields () [virtual slot]

Causes the default form to write its fields to the record buffer. If there is no default form, or no record, nothing happens.

See also setForm() [p. 63].

QSql Class Reference

The QSql class is a namespace for Qt SQL identifiers that need to be global-like.

This class is part of the **sql** module.

```
#include <qsql.h>
```

Public Members

- **QSql**()
- enum **Op** { None = -1, Insert = 0, Update = 1, Delete = 2 }
- enum **Location** { BeforeFirst = -1, AfterLast = -2 }
- enum **Confirm** { Cancel = -1, No = 0, Yes = 1 }

Detailed Description

The QSql class is a namespace for Qt SQL identifiers that need to be global-like.

Normally, you can ignore this class. Several Qt SQL classes inherit it, so all the identifiers in the Qt SQL namespace are visible to you without qualification.

See also Database Classes.

Member Type Documentation

QSql::Confirm

This enum type describes edit confirmations.

The currently defined values are:

- `QSql::Yes`
- `QSql::No`
- `QSql::Cancel`

QSql::Location

This enum type describes SQL navigation locations.

The currently defined values are:

- `QSql::BeforeFirst`
- `QSql::AfterLast`

QSql::Op

This enum type describes edit operations.

The currently defined values are:

- `QSql::None`
- `QSql::Insert`
- `QSql::Update`
- `QSql::Delete`

Member Function Documentation

QSql::QSql ()

Constructs a Qt SQL namepsace class

QSqlCursor Class Reference

The QSqlCursor class provides browsing and editing of SQL tables and views.

This class is part of the **sql** module.

`#include <qsqlcursor.h>`

Inherits QSqlRecord [p. 122] and QSqlQuery [p. 115].

Public Members

- **QSqlCursor** (const QString & name = QString::null, bool autopopulate = TRUE, QSqlDatabase * db = 0)
- **QSqlCursor** (const QSqlCursor & other)
- QSqlCursor & **operator=** (const QSqlCursor & other)
- **~QSqlCursor** ()
- enum **Mode** { ReadOnly = 0, Insert = 1, Update = 2, Delete = 4, Writable = 7 }
- virtual QSqlIndex **primaryIndex** (bool setFromCursor = TRUE) const
- virtual QSqlIndex **index** (const QStringList & fieldNames) const
- QSqlIndex **index** (const QString & fieldName) const
- QSqlIndex **index** (const char * fieldName) const
- virtual void **setPrimaryIndex** (const QSqlIndex & idx)
- virtual void **append** (const QSqlFieldInfo & fieldInfo)
- virtual void **insert** (int pos, const QSqlFieldInfo & fieldInfo)
- virtual void **remove** (int pos)
- virtual void **clear** ()
- virtual void **setGenerated** (const QString & name, bool generated)
- virtual void **setGenerated** (int i, bool generated)
- virtual QSqlRecord * **editBuffer** (bool copy = FALSE)
- virtual QSqlRecord * **primeInsert** ()
- virtual QSqlRecord * **primeUpdate** ()
- virtual QSqlRecord * **primeDelete** ()
- virtual int **insert** (bool invalidate = TRUE)
- virtual int **update** (bool invalidate = TRUE)
- virtual int **del** (bool invalidate = TRUE)
- virtual void **setMode** (int mode)
- int **mode** () const
- virtual void **setCalculated** (const QString & name, bool calculated)
- bool **isCalculated** (const QString & name) const
- virtual void **setTrimmed** (const QString & name, bool trim)
- bool **isTrimmed** (const QString & name) const
- bool **isReadOnly** () const

- bool **canInsert** () const
- bool **canUpdate** () const
- bool **canDelete** () const
- bool **select** ()
- bool **select** (const QSqlIndex & sort)
- bool **select** (const QSqlIndex & filter, const QSqlIndex & sort)
- virtual bool **select** (const QString & filter, const QSqlIndex & sort = QSqlIndex ( ))
- virtual void **setSort** (const QSqlIndex & sort)
- QSqlIndex **sort** () const
- virtual void **setFilter** (const QString & filter)
- QString **filter** () const
- virtual void **setName** (const QString & name, bool autopopulate = TRUE)
- QString **name** () const

Protected Members

- virtual QVariant **calculateField** (const QString & name)
- virtual int **update** (const QString & filter, bool invalidate = TRUE)
- virtual int **del** (const QString & filter, bool invalidate = TRUE)
- virtual QString **toString** (const QString & prefix, QSqlField * field, const QString & fieldSep) const
- virtual QString **toString** (QSqlRecord * rec, const QString & prefix, const QString & fieldSep, const QString & sep) const
- virtual QString **toString** (const QSqlIndex & i, QSqlRecord * rec, const QString & prefix, const QString & fieldSep, const QString & sep) const

Detailed Description

The QSqlCursor class provides browsing and editing of SQL tables and views.

A QSqlCursor is a database record (see QSqlRecord) that corresponds to a table or view within an SQL database (see QSqlDatabase). There are two buffers in a cursor, one used for browsing and one used for editing records. Each buffer contains a list of fields which correspond to the fields in the table or view.

When positioned on a valid record, the browse buffer contains the values of the current record's fields from the database. The edit buffer is separate, and is used for editing existing records and inserting new records.

For browsing data, a cursor must first select() data from the database. After a successful select() the cursor is active (isActive() returns TRUE), but is initially not positioned on a valid record (isValid() returns FALSE). To position the cursor on a valid record, use one of the navigation functions, next(), prev(), first(), last(), or seek(). Once positioned on a valid record, data can be retrieved from the browse buffer using value(). If a navigation function is not successful, it returns FALSE, the cursor will no longer be positioned on a valid record and the values returned by value() are undefined.

For example:

```
QSqlCursor cur( "staff" ); // Specify the table/view name
cur.select(); // We'll retrieve every record
while ( cur.next() ) {
    qDebug( cur.value( "id" ).toString() + ": " +
            cur.value( "surname" ).toString() + " " +
            cur.value( "salary" ).toString() );
}
```

In the above example, a cursor is created specifying a table or view name in the database. Then, select() is called, which can be optionally parameterised to filter and order the records retrieved. Each record in the cursor is retrieved using next(). When next() returns FALSE, there are no more records to process, and the loop terminates.

For editing records (rows of data), a cursor contains a separate edit buffer which is independent of the fields used when browsing. The functions insert(), update() and del() operate on the edit buffer. This allows the cursor to be repositioned to other records while simultaneously maintaining a separate buffer for edits. You can get a pointer to the edit buffer using editBuffer(). The primeInsert(), primeUpdate() and primeDelete() functions also return a pointer to the edit buffer and prepare it for insert, update and delete respectively. Edit operations only affect a single row at a time. Note that update() and del() require that the table or view contain a primaryIndex() to ensure that edit operations affect a unique record within the database.

For example:

```
QSqlCursor cur( "prices" );
cur.select( "id=202" );
if ( cur.next() ) {
    QSqlRecord *buffer = cur.primeUpdate();
    double price = buffer->value( "price" ).toDouble();
    double newprice = price * 1.05;
    buffer->setValue( "price", newprice );
    cur.update();
}
```

To edit an existing database record, first move to the record you wish to update. Call primeUpdate() to get the pointer to the cursor's edit buffer. Then use this pointer to modify the values in the edit buffer. Finally, call update() to save the changes to the database. The values in the edit buffer will be used to locate the appropriate record when updating the database (see primaryIndex()).

Similarly, when deleting an existing database record, first move the record you wish to delete. Then, call primeDelete() to get the pointer to the edit buffer. Finally, call del() to delete the record from the database. Again, the values in the edit buffer will be used to locate and delete the appropriate record.

To insert a new record, call primeInsert() to get the pointer to the edit buffer. Use this pointer to populate the edit buffer with new values and then insert() the record into the database.

After calling insert(), update() or del(), the cursor is no longer positioned on a valid record and can no longer be navigated (isValid() return FALSE). The reason for this is that any changes made to the database will not be visible until select() is called to refresh the cursor. You can change this behavior by passing FALSE to insert(), update() or del() which will prevent the cursor from becoming invalid. These edits will then not be visible when navigating the cursor until select() is called.

QSqlCursor contains virtual methods which allow editing behavior to be customized by subclasses. This allows custom cursors to be created which encapsulate the editing behavior of a database table for an entire application. For example, a cursor can be customized to always auto-number primary index fields, or provide fields with suitable default values, when inserting new records.

See also Database Classes.

Member Type Documentation

QSqlCursor::Mode

This enum type describes how QSqlCursor operates on records in the database.

The currently defined values are:

- `QSqlCursor::ReadOnly` - the cursor can only SELECT records from the database.

- `QSqlCursor::Insert` - the cursor can INSERT records into the database.
- `QSqlCursor::Update` - the cursor can UPDATE records in the database.
- `QSqlCursor::Delete` - the cursor can DELETE records from the database.
- `QSqlCursor::Writable` - the cursor can INSERT, UPDATE and DELETE records in the database.

Member Function Documentation

QSqlCursor::QSqlCursor (const QString & name = QString::null, bool autopopulate = TRUE, QSqlDatabase * db = 0)

Constructs a cursor on database *db* using table or view *name*.

If *autopopulate* is TRUE (the default), the *name* of the cursor must correspond to an existing table or view name in the database so that field information can be automatically created. If the table or view does not exist, the cursor will not be functional.

The cursor is created with an initial mode of QSqlCursor::Writable (meaning that records can be inserted, updated or deleted using the cursor). If the cursor does not have a unique primary index, update and deletes cannot be performed.

Note that *autopopulate* refers to populating the cursor with meta-data, e.g. the names of the table's fields, not with retrieving data. The select() function is used to populate the cursor with data.

See also setName() [p. 77] and setMode() [p. 76].

QSqlCursor::QSqlCursor (const QSqlCursor & other)

Constructs a copy of *other*.

QSqlCursor::~QSqlCursor ()

Destroys the object and frees any allocated resources.

void QSqlCursor::append (const QSqlFieldInfo & fieldInfo) [virtual]

Append a copy of field *fieldInfo* to the end of the cursor. Note that all references to the cursor edit buffer become invalidated.

QVariant QSqlCursor::calculateField (const QString & name) [virtual protected]

Protected virtual function which is called whenever a field needs to be calculated. If calculated fields are being used, derived classes must reimplement this function and return the appropriate value for field *name*. The default implementation returns an invalid QVariant.

See also setCalculated() [p. 76].

Examples: sql/overview/subclass3/main.cpp and sql/overview/subclass4/main.cpp.

bool QSqlCursor::canDelete () const

Returns TRUE if the cursor will perform deletes, FALSE otherwise.

See also setMode() [p. 76].

bool QSqlCursor::canInsert () const

Returns TRUE if the cursor will perform inserts, FALSE otherwise.

See also setMode() [p. 76].

bool QSqlCursor::canUpdate () const

Returns TRUE if the cursor will perform updates, FALSE otherwise.

See also setMode() [p. 76].

void QSqlCursor::clear () [virtual]

Removes all fields from the cursor. Note that all references to the cursor edit buffer become invalidated.

Reimplemented from QSqlRecord [p. 123].

int QSqlCursor::del (bool invalidate = TRUE) [virtual]

Deletes a record from the database using the cursor's primary index and the contents of the cursor edit buffer. Returns the number of records which were deleted, or 0 if there was an error. For error information, use lastError().

Only records which meet the filter criteria specified by the cursor's primary index are deleted. If the cursor does not contain a primary index, no delete is performed and 0 is returned. If *invalidate* is TRUE (the default), the current cursor can no longer be navigated. A new select() call must be made before you can move to a valid record. For example:

```
QSqlCursor cur( "prices" );
cur.select( "id=999" );
if ( cur.next() ) {
    cur.primeDelete();
    cur.del();
}
```

In the above example, a cursor is created on the 'prices' table and positioned to the record to be deleted. First primeDelete() is called to populate the edit buffer with the current cursor values, e.g. with an id of 999, and then del() is called to actually delete the record from the database. Remember: all edit operations (insert(), update() and delete()) operate on the contents of the cursor edit buffer and not on the contents of the cursor itself.

See also primeDelete() [p. 74], setMode() [p. 76] and lastError() [p. 118].

Example: sql/overview/del/main.cpp.

int QSqlCursor::del (const QString & filter, bool invalidate = TRUE) [virtual protected]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Deletes the current cursor record from the database using the filter *filter*. Only records which meet the filter criteria are deleted. Returns the number of records which were deleted. If *invalidate* is TRUE (the default), the current cursor can no longer be navigated. A new select() call must be made before you can move to a valid record. For error information, use lastError().

The *filter* is an SQL `WHERE` clause, e.g. `id=500`.

See also setMode() [p. 76] and lastError() [p. 118].

QSqlRecord * QSqlCursor::editBuffer (bool copy = FALSE) [virtual]

Returns a pointer to the current internal edit buffer. If *copy* is TRUE (the default is FALSE), the current cursor field values are first copied into the edit buffer. The edit buffer is valid as long as the cursor remains valid. The cursor retains ownership of the returned pointer, so it must not be deleted or modified.

See also primeInsert() [p. 74], primeUpdate() [p. 74] and primeDelete() [p. 74].

QString QSqlCursor::filter () const

Returns the current filter, or an empty string if there is no current filter.

QSqlIndex QSqlCursor::index (const QStringList & fieldNames) const [virtual]

Returns an index composed of *fieldNames*, all in ASCending order. Note that all field names must exist in the cursor, otherwise an empty index is returned.

See also QSqlIndex [p. 109].

Examples: sql/overview/extract/main.cpp, sql/overview/order1/main.cpp, sql/overview/order2/main.cpp and sql/overview/table3/main.cpp.

QSqlIndex QSqlCursor::index (const QString & fieldName) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns an index based on *fieldName*.

QSqlIndex QSqlCursor::index (const char * fieldName) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns an index based on *fieldName*.

void QSqlCursor::insert (int pos, const QSqlFieldInfo & fieldInfo) [virtual]

Insert a copy of *fieldInfo* at position *pos*. If a field already exists at *pos*, it is removed. Note that all references to the cursor edit buffer become invalidated.

Examples: sql/overview/insert/main.cpp and sql/overview/insert2/main.cpp.

int QSqlCursor::insert (bool invalidate = TRUE) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Inserts the current contents of the cursor's edit record buffer into the database, if the cursor allows inserts. Returns the number of rows affected by the insert. For error information, use lastError().

If *invalidate* is TRUE (the default), the cursor will no longer be positioned on a valid record and can no longer be navigated. A new select() call must be made before navigating to a valid record.

```
QSqlCursor cur( "prices" );
QSqlRecord *buffer = cur.primeInsert();
buffer->setValue( "id",    53981 );
buffer->setValue( "name",  "Thingy" );
buffer->setValue( "price", 105.75 );
cur.insert();
```

In the above example, a cursor is created on the 'prices' table and a pointer to the insert buffer is aquired using primeInsert(). Each field's value is set to the desired value and then insert() is called to insert the data into the database. Remember: all edit operations (insert(), update() and delete()) operate on the contents of the cursor edit buffer and not on the contents of the cursor itself.

See also setMode() [p. 76] and lastError() [p. 118].

bool QSqlCursor::isCalculated (const QString & name) const

Returns TRUE if the field *name* is calculated, otherwise FALSE is returned. If the field *name* does not exist, FALSE is returned.

See also setCalculated() [p. 76].

bool QSqlCursor::isReadOnly () const

Returns TRUE if the cursor is read-only, FALSE otherwise. The default is FALSE. Read-only cursors cannot be edited using insert(), update() or del().

See also setMode() [p. 76].

bool QSqlCursor::isTrimmed (const QString & name) const

Returns TRUE if the field *name* is trimmed, otherwise FALSE is returned. If the field *name* does not exist, FALSE is returned.

When a trimmed field of type string or cstring is read from the database any trailing (right-most) spaces are removed.

See also setTrimmed() [p. 77].

int QSqlCursor::mode () const

Returns the current cursor mode.

See also setMode() [p. 76].

QString QSqlCursor::name () const

Returns the name of the cursor.

QSqlCursor & QSqlCursor::operator= (const QSqlCursor & other)

Sets the cursor equal to *other*.

QSqlIndex QSqlCursor::primaryIndex (bool setFromCursor = TRUE) const [virtual]

Returns the primary index associated with the cursor as defined in the database, or an empty index if there is no primary index. If *setFromCursor* is TRUE (the default), the index fields are populated with the corresponding values in the cursor's current record.

QSqlRecord * QSqlCursor::primeDelete () [virtual]

'Primes' the field values of the edit buffer for delete and returns a pointer to the edit buffer. The default implementation copies the field values from the current cursor record into the edit buffer (therefore, this function is equivalent to calling editBuffer(TRUE)). The cursor retains ownership of the returned pointer, so it must not be deleted or modified.

See also editBuffer() [p. 72] and del() [p. 71].

Example: sql/overview/del/main.cpp.

QSqlRecord * QSqlCursor::primeInsert () [virtual]

'Primes' the field values of the edit buffer for insert and returns a pointer to the edit buffer. The default implementation clears all field values in the edit buffer. The cursor retains ownership of the returned pointer, so it must not be deleted or modified.

See also editBuffer() [p. 72] and insert() [p. 72].

Examples: sql/overview/insert/main.cpp, sql/overview/insert2/main.cpp, sql/overview/subclass5/main.cpp and sql/sqltable/main.cpp.

QSqlRecord * QSqlCursor::primeUpdate () [virtual]

'Primes' the field values of the edit buffer for update and returns a pointer to the edit buffer. The default implementation copies the field values from the current cursor record into the edit buffer (therefore, this function is equivalent to calling editBuffer(TRUE)). The cursor retains ownership of the returned pointer, so it must not be deleted or modified.

See also editBuffer() [p. 72] and update() [p. 78].

Examples: sql/overview/custom1/main.cpp, sql/overview/form1/main.cpp and sql/overview/update/main.cpp.

void QSqlCursor::remove (int pos) [virtual]

Removes the field at *pos*. If *pos* does not exist, nothing happens. Note that all references to the cursor edit buffer become invalidated.

Reimplemented from QSqlRecord [p. 125].

bool QSqlCursor::select (const QString & filter, const QSqlIndex & sort = QSqlIndex ()) [virtual]

Selects all fields in the cursor from the database matching the filter criteria *filter*. The data is returned in the order specified by the index *sort*. Returns TRUE if the data was successfully selected, otherwise FALSE is returned.

The *filter* is a string containing an SQL `WHERE` clause but without the 'WHERE' keyword. The cursor is initially positioned at an invalid row after this function is called. To move to a valid row, use seek(), first(), last(), prev() or next().

Example:

```
QSqlCursor cur( "Employee" ); // Use the Employee table or view
cur.select( "deptno=10" ); // select all records in department 10
while( cur.next() ) {
    ... // process data
}
...
// select records in other departments, ordered by department number
cur.select( "deptno>10", cur.index( "deptno" ) );
...
```

The filter will apply to any subsequent select() calls that do not explicitly specify another filter. Similarly the sort will apply to any subsequent select() calls that do not explicitly specify another sort.

```
QSqlCursor cur( "Employee" );
cur.select( "deptno=10" ); // select all records in department 10
while( cur.next() ) {
    ... // process data
}
...
cur.select(); // re-selects all records in department 10
...
```

Examples: sql/overview/del/main.cpp, sql/overview/extract/main.cpp, sql/overview/order1/main.cpp, sql/overview/order2/main.cpp, sql/overview/retrieve2/main.cpp, sql/overview/table3/main.cpp and sql/overview/update/main.cpp.

bool QSqlCursor::select ()

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Selects all fields in the cursor from the database. The rows are returned in the order specified by the last call to setSort() or the last call to select() that specified a sort, whichever is the most recent. If there is no current sort, the order in which the rows are returned is undefined. The records are filtered according to the filter specified by the last call to setFilter() or the last call to select() that specified a filter, whichever is the most recent. If there is no current filter, all records are returned. The cursor is initially positioned at an invalid row. To move to a valid row, use seek(), first(), last(), prev() or next().

See also setSort() [p. 77] and setFilter() [p. 76].

bool QSqlCursor::select (const QSqlIndex & sort)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Selects all fields in the cursor from the database. The data is returned in the order specified by the index *sort*. The records are filtered according to the filter specified by the last call to setFilter() or the last call to select() that specified a filter, whichever is the most recent. The cursor is initially positioned at an invalid row. To move to a valid row, use seek(), first(), last(), prev() or next().

bool QSqlCursor::select (const QSqlIndex & filter, const QSqlIndex & sort)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Selects all fields in the cursor matching the filter index *filter*. The data is returned in the order specified by the index *sort*. The *filter* index works by constructing a WHERE clause using the names of the fields from the *filter* and

their values from the current cursor record. The cursor is initially positioned at an invalid row. To move to a valid row, use seek(), first(), last(), prev() or next(). This function is useful, for example, for retrieving data based upon a table's primary index:

```
QSqlCursor cur( "Employee" );
QSqlIndex pk = cur.primaryIndex();
cur.setValue( "id", 10 );
cur.select( pk, pk ); // generates "SELECT ... FROM Employee WHERE id=10 ORDER BY id"
...
```

In this example the QSqlIndex, pk, is used for two different purposes. When used as the filter (first) argument, the field names it contains are used to construct the WHERE clause, each set to the current cursor value, `WHERE id=10`, in this case. When used as the sort (second) argument the field names it contains are used for the ORDER BY clause, `ORDER BY id` in this example.

void QSqlCursor::setCalculated (const QString & name, bool calculated) [virtual]

Sets field *name* to *calculated*. If the field *name* does not exist, nothing happens. The value of a calculated field is set by the calculateField() virtual function which you must reimplement otherwise the field value will become an invalid QVariant. Calculated fields do not appear in generated SQL statements sent to the database.

See also calculateField() [p. 70] and QSqlRecord::setGenerated() [p. 125].

void QSqlCursor::setFilter (const QString & filter) [virtual]

Sets the current filter to *filter*. Note that no new records are selected. To select new records, use select(). The *filter* will apply to any subsequent select() calls that do not explicitly specify a filter.

The filter is an SQL `WHERE` clause without the keyword 'WHERE', e.g. `name='Dave'`.

void QSqlCursor::setGenerated (const QString & name, bool generated) [virtual]

Sets the generated flag for the field *name* to *generated*. If the field does not exist, nothing happens. Only fields that have *generated* set to TRUE are included in the SQL that is generated, e.g. by QSqlCursor.

See also isGenerated() [p. 124].

Reimplemented from QSqlRecord [p. 125].

void QSqlCursor::setGenerated (int i, bool generated) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the generated flag for the field *i* to *generated*.

See also isGenerated() [p. 124].

Reimplemented from QSqlRecord [p. 125].

void QSqlCursor::setMode (int mode) [virtual]

Sets the cursor mode to *mode*. This value can be an OR'ed combination of QSqlCursor::Mode values. The default mode for a cursor is QSqlCursor::Writable.

```
QSqlCursor cur( "Employee" );
cur.setMode( QSqlCursor::Writable ); // allow insert/update/delete
...
cur.setMode( QSqlCursor::Insert | QSqlCursor::Update ); // allow inserts and updates only
...
cur.setMode( QSqlCursor::ReadOnly ); // no inserts/updates/deletes allowed
```

void QSqlCursor::setName (const QString & name, bool autopopulate = TRUE) [virtual]

Sets the name of the cursor to *name*. If *autopopulate* is TRUE (the default), the *name* must correspond to a valid table or view name in the database. Also, note that all references to the cursor edit buffer become invalidated when fields are auto-populated. See the QSqlCursor constructor documentation for more information.

void QSqlCursor::setPrimaryIndex (const QSqlIndex & idx) [virtual]

Sets the primary index associated with the cursor to the index *idx*. Note that this index must contain a field or set of fields which identify a unique record within the underlying database table or view so that update() and del() will execute as expected.

See also update() [p. 78] and del() [p. 71].

void QSqlCursor::setSort (const QSqlIndex & sort) [virtual]

Sets the current sort to *sort*. Note that no new records are selected. To select new records, use select(). The *sort* will apply to any subsequent select() calls that do not explicitly specify a sort.

void QSqlCursor::setTrimmed (const QString & name, bool trim) [virtual]

Sets field *name* to *trim*. If the field *name* does not exist, nothing happens.

When a trimmed field of type string or cstring is read from the database any trailing (right-most) spaces are removed.

See QVariant.

See also isTrimmed() [p. 73].

QSqlIndex QSqlCursor::sort () const

Returns the current sort, or an empty index if there is no current sort.

QString QSqlCursor::toString (QSqlRecord * rec, const QString & prefix, const QString & fieldSep, const QString & sep) const [virtual protected]

Returns a formatted string composed of all the fields in *rec*. Each field is composed of the *prefix* (e.g. table or view name), ".", the field name, the *fieldSep* and the field value. If the *prefix* is empty then the field will begin with the field name. The fields are then joined together separated by *sep*. Fields where isGenerated() returns FALSE are not included. This function is useful for generating SQL statements.

QString QSqlCursor::toString (const QString & prefix, QSqlField * field, const QString & fieldSep) const [virtual protected]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a formatted string composed of the *prefix* (e.g. table or view name), ".", the *field* name, the *fieldSep* and the field value. If the *prefix* is empty then the string will begin with the *field* name. This function is useful for generating SQL statements.

QString QSqlCursor::toString (const QSqlIndex & i, QSqlRecord * rec, const QString & prefix, const QString & fieldSep, const QString & sep) const [virtual protected]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a formatted string composed of all the fields in the index *i*. Each field is composed of the *prefix* (e.g. table or view name), ".", the field name, the *fieldSep* and the field value. If the *prefix* is empty then the field will begin with the field name. The field values are taken from *rec*. The fields are then joined together separated by *sep*. Fields where isGenerated() returns FALSE are ignored. This function is useful for generating SQL statements.

int QSqlCursor::update (bool invalidate = TRUE) [virtual]

Updates the database with the current contents of the edit buffer. Returns the number of records which were updated, or 0 if there was an error. For error information, use lastError().

Only records which meet the filter criteria specified by the cursor's primary index are updated. If the cursor does not contain a primary index, no update is performed and 0 is returned.

If *invalidate* is TRUE (the default), the current cursor can no longer be navigated. A new select() call must be made before you can move to a valid record. For example:

```
QSqlCursor cur( "prices" );
cur.select( "id=202" );
if ( cur.next() ) {
    QSqlRecord *buffer = cur.primeUpdate();
    double price = buffer->value( "price" ).toDouble();
    double newprice = price * 1.05;
    buffer->setValue( "price", newprice );
    cur.update();
}
```

In the above example, a cursor is created on the 'prices' table and is positioned on the record to be update updated. A pointer is then aquired to the cursor's edit buffer using primeUpdate(). A new value is calculated and placed into the edit buffer with the setValue() call. Finally, an update() call is made on the cursor which uses the tables's primary index to update the record in the database with the contents of the cursor's edit buffer. Remember: all edit operations (insert(), update() and delete()) operate on the contents of the cursor edit buffer and not on the contents of the cursor itself.

Note that if the primary index does not uniquely distinguish records the database may be changed into an inconsistent state.

See also setMode() [p. 76] and lastError() [p. 118].

Example: sql/overview/update/main.cpp.

int QSqlCursor::update (const QString & filter, bool invalidate = TRUE) [virtual protected]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Updates the database with the current contents of the cursor edit buffer using the specified *filter*. Returns the number of records which were updated, or 0 if there was an error. For error information, use lastError().

Only records which meet the filter criteria are updated, otherwise all records in the table are updated.

If *invalidate* is TRUE (the default), the cursor can no longer be navigated. A new select() call must be made before you can move to a valid record.

See also primeUpdate() [p. 74], setMode() [p. 76] and lastError() [p. 118].

QSqlDatabase Class Reference

The QSqlDatabase class is used to create SQL database connections and provide transaction handling.

This class is part of the **sql** module.

`#include <qsqldatabase.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **~QSqlDatabase** ()
- bool **open** ()
- bool **open** (const QString & user, const QString & password)
- void **close** ()
- bool **isOpen** () const
- bool **isOpenError** () const
- QStringList **tables** () const
- QSqlIndex **primaryIndex** (const QString & tablename) const
- QSqlRecord **record** (const QString & tablename) const
- QSqlRecord **record** (const QSqlQuery & query) const
- QSqlRecordInfo **recordInfo** (const QString & tablename) const
- QSqlRecordInfo **recordInfo** (const QSqlQuery & query) const
- QSqlQuery **exec** (const QString & query = QString::null) const
- QSqlError **lastError** () const
- bool **transaction** ()
- bool **commit** ()
- bool **rollback** ()
- virtual void **setDatabaseName** (const QString & name)
- virtual void **setUserName** (const QString & name)
- virtual void **setPassword** (const QString & password)
- virtual void **setHostName** (const QString & host)
- virtual void **setPort** (int p)
- QString **databaseName** () const
- QString **userName** () const
- QString **password** () const
- QString **hostName** () const
- QString **driverName** () const
- int **port** () const
- QSqlDriver * **driver** () const

Static Public Members

- QSqlDatabase * **addDatabase** (const QString & type, const QString & connectionName = defaultConnection)
- QSqlDatabase * **database** (const QString & connectionName = defaultConnection, bool open = TRUE)
- void **removeDatabase** (const QString & connectionName)
- bool **contains** (const QString & connectionName = defaultConnection)
- QStringList **drivers** ()

Properties

- QString **databaseName** — the name of the database or the TNS Service Name for the QOCI8 (Oracle) driver
- QString **hostName** — the host name where the database resides
- QString **password** — the password used to connect to the database
- int **port** — the port used to connect to the database
- QString **userName** — the user name connected to the database

Protected Members

- **QSqlDatabase** (const QString & driver, const QString & name, QObject * parent = 0, const char * objname = 0)

Detailed Description

The QSqlDatabase class is used to create SQL database connections and provide transaction handling.

This class is used to create connections to SQL databases. It also provides transaction handling functions for those database drivers that support transactions.

The QSqlDatabase class itself provides an abstract interface for accessing many types of database backend. Database-specific drivers are used internally to actually access and manipulate data, (see QSqlDriver). Result set objects provide the interface for executing and manipulating SQL queries (see QSqlQuery).

See also Database Classes.

Member Function Documentation

QSqlDatabase::QSqlDatabase (const QString & driver, const QString & name, QObject * parent = 0, const char * objname = 0) [protected]

Creates a QSqlDatabase connection named *name* that uses the driver referred to by *driver*, with the parent *parent* and the object name *objname*. If the *driver* is not recognized, the database connection will have no functionality.

The currently available drivers are:

- QODBC3 - ODBC (Open Database Connectivity) Driver
- QOCI8 - Oracle Call Interface Driver
- QPSQL7 - PostgreSQL v6.x and v7.x Driver
- QTDS7 - Sybase Adaptive Server and Microsoft SQL Server Driver

- QMYSQL3 - MySQL Driver

Note that additional 3rd party drivers can be loaded dynamically.

QSqlDatabase::~QSqlDatabase ()

Destroys the object and frees any allocated resources.

QSqlDatabase * QSqlDatabase::addDatabase (const QString & type, const QString & connectionName = defaultConnection) [static]

Adds a database to the list of database connections using the driver *type* and the connection name *connectionName*.

The database connection is referred to by *connectionName*. A pointer to the newly added database connection is returned. This pointer is owned by QSqlDatabase and will be deleted on program exit or when removeDatabase() is called. If *connectionName* is not specified, the newly added database connection becomes the default database connection for the application, and subsequent calls to database() (without a database name parameter) will return a pointer to it.

See also database() [p. 82] and removeDatabase() [p. 85].

Examples: sql/overview/connect1/main.cpp, sql/overview/create_connections/main.cpp and sql/sqltable/main.cpp.

void QSqlDatabase::close ()

Closes the database connection, freeing any resources acquired.

bool QSqlDatabase::commit ()

Commits a transaction to the database if the driver supports transactions. Returns TRUE if the operation succeeded, FALSE otherwise.

See also QSqlDriver::hasFeature() [p. 89] and rollback() [p. 85].

bool QSqlDatabase::contains (const QString & connectionName = defaultConnection) [static]

Returns TRUE if the list of database connections contains *connectionName*, otherwise returns FALSE.

QSqlDatabase * QSqlDatabase::database (const QString & connectionName = defaultConnection, bool open = TRUE) [static]

Returns a pointer to the database connection named *connectionName*. The database connection must have been previously added with database(). If *open* is TRUE (the default) and the database connection is not already open it is opened now. If no *connectionName* is specified the default connection is used. If *connectionName* does not exist in the list of databases, 0 is returned. The pointer returned is owned by QSqlDatabase and should *not* be deleted.

Examples: sql/overview/basicbrowsing/main.cpp and sql/overview/create_connections/main.cpp.

QString QSqlDatabase::databaseName () const

Returns the name of the database or the TNS Service Name for the QOCI8 (Oracle) driver. See the "databaseName" [p. 86] property for details.

QSqlDriver * QSqlDatabase::driver () const

Returns a pointer to the database driver used to access the database connection.

QString QSqlDatabase::driverName () const

Returns the name of the driver used by the database connection.

QStringList QSqlDatabase::drivers () [static]

Returns a list of all available database drivers.

QSqlQuery QSqlDatabase::exec (const QString & query = QString::null) const

Executes an SQL statement (e.g. an INSERT, UPDATE or DELETE statement) on the database, and returns a QSqlQuery object. Use lastError() to retrieve error information. If *query* is QString::null, an empty, invalid query is returned and lastError() is not affected.

See also QSqlQuery [p. 115] and lastError() [p. 83].

QString QSqlDatabase::hostName () const

Returns the host name where the database resides. See the "hostName" [p. 86] property for details.

bool QSqlDatabase::isOpen () const

Returns TRUE if the database connection is currently open, otherwise returns FALSE.

bool QSqlDatabase::isOpenError () const

Returns TRUE if there was an error opening the database connection, otherwise returns FALSE. Error information can be retrieved using the lastError() function.

QSqlError QSqlDatabase::lastError () const

Returns information about the last error that occurred on the database. See QSqlError for more information.

Examples: sql/overview/create_connections/main.cpp and sql/sqltable/main.cpp.

bool QSqlDatabase::open ()

Opens the database connection using the current connection values. Returns TRUE on success, and FALSE if there was an error. Error information can be retrieved using the lastError() function.

See also lastError() [p. 83].

Examples: sql/overview/connect1/main.cpp, sql/overview/create_connections/main.cpp and sql/sqltable/main.cpp.

bool QSqlDatabase::open (const QString & user, const QString & password)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Opens the database connection using *user* name and *password*. Returns TRUE on success, and FALSE if there was an error. Error information can be retrieved using the lastError() function.

See also lastError() [p. 83].

QString QSqlDatabase::password () const

Returns the password used to connect to the database. See the "password" [p. 86] property for details.

int QSqlDatabase::port () const

Returns the port used to connect to the database. See the "port" [p. 86] property for details.

QSqlIndex QSqlDatabase::primaryIndex (const QString & tablename) const

Returns the primary index for table *tablename*. If no primary index exists an empty QSqlIndex will be returned.

QSqlRecord QSqlDatabase::record (const QString & tablename) const

Returns a QSqlRecord populated with the names of all the fields in the table (or view) named *tablename*. The order in which the fields are returned is undefined. If no such table (or view) exists, an empty record is returned.

See also recordInfo() [p. 84].

QSqlRecord QSqlDatabase::record (const QSqlQuery & query) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a QSqlRecord populated with the names of all the fields used in the SQL *query*. If the query is a "SELECT *" the order in which fields are returned is undefined.

See also recordInfo() [p. 84].

QSqlRecordInfo QSqlDatabase::recordInfo (const QString & tablename) const

Returns a QSqlRecordInfo populated with meta-data about the table (or view) *tablename*. If no such table (or view) exists, an empty record is returned.

See also QSqlRecordInfo [p. 128], QSqlFieldInfo [p. 101] and record() [p. 84].

QSqlRecordInfo QSqlDatabase::recordInfo (const QSqlQuery & query) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a QSqlRecordInfo object with meta data for the QSqlQuery *query*. Note that this overloaded function may return not as much information as the recordInfo function which takes the name of a table as parameter.

See also QSqlRecordInfo [p. 128], QSqlFieldInfo [p. 101] and record() [p. 84].

void QSqlDatabase::removeDatabase (const QString & connectionName) [static]

Removes the database connection *connectionName* from the list of database connections. Note that there should be no open queries on the database connection when this function is called, otherwise a resource leak will occur.

bool QSqlDatabase::rollback ()

Rolls a transaction back on the database if the driver supports transactions. Returns TRUE if the operation succeeded, FALSE otherwise.

See also QSqlDriver::hasFeature() [p. 89], commit() [p. 82] and transaction() [p. 85].

void QSqlDatabase::setDatabaseName (const QString & name) [virtual]

Sets the name of the database or the TNS Service Name for the QOCI8 (Oracle) driver to *name*. See the "databaseName" [p. 86] property for details.

void QSqlDatabase::setHostName (const QString & host) [virtual]

Sets the host name where the database resides to *host*. See the "hostName" [p. 86] property for details.

void QSqlDatabase::setPassword (const QString & password) [virtual]

Sets the password used to connect to the database to *password*. See the "password" [p. 86] property for details.

void QSqlDatabase::setPort (int p) [virtual]

Sets the port used to connect to the database to *p*. See the "port" [p. 86] property for details.

void QSqlDatabase::setUserName (const QString & name) [virtual]

Sets the user name connected to the database to *name*. See the "userName" [p. 86] property for details.

QStringList QSqlDatabase::tables () const

Returns a list of tables in the database.

bool QSqlDatabase::transaction ()

Begins a transaction on the database if the driver supports transactions. Returns TRUE if the operation succeeded, FALSE otherwise.

See also QSqlDriver::hasFeature() [p. 89], commit() [p. 82] and rollback() [p. 85].

QString QSqlDatabase::userName () const

Returns the user name connected to the database. See the "userName" [p. 86] property for details.

Property Documentation

QString databaseName

This property holds the name of the database or the TNS Service Name for the QOCI8 (Oracle) driver.

There is no default value.

Set this property's value with setDatabaseName() and get this property's value with databaseName().

QString hostName

This property holds the host name where the database resides.

There is no default value.

Set this property's value with setHostName() and get this property's value with hostName().

QString password

This property holds the password used to connect to the database.

There is no default value.

Set this property's value with setPassword() and get this property's value with password().

int port

This property holds the port used to connect to the database.

There is no default value.

Set this property's value with setPort() and get this property's value with port().

QString userName

This property holds the user name connected to the database.

There is no default value.

Set this property's value with setUserName() and get this property's value with userName().

QSqlDriver Class Reference

The QSqlDriver class is an abstract base class for accessing SQL databases.

This class is part of the **sql** module.

`#include <qsqldriver.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- enum **DriverFeature** { Transactions, QuerySize, BLOB }
- **QSqlDriver** (QObject * parent = 0, const char * name = 0)
- **~QSqlDriver** ()
- bool **isOpen** () const
- bool **isOpenError** () const
- virtual bool **beginTransaction** ()
- virtual bool **commitTransaction** ()
- virtual bool **rollbackTransaction** ()
- virtual QStringList **tables** (const QString & user) const
- virtual QSqlIndex **primaryIndex** (const QString & tableName) const
- virtual QSqlRecord **record** (const QString & tableName) const
- virtual QSqlRecord **record** (const QSqlQuery & query) const
- virtual QSqlRecordInfo **recordInfo** (const QString & tablename) const
- virtual QSqlRecordInfo **recordInfo** (const QSqlQuery & query) const
- virtual QString **nullText** () const
- virtual QString **formatValue** (const QSqlField * field, bool trimStrings = FALSE) const
- QSqlError **lastError** () const
- virtual bool **hasFeature** (DriverFeature f) const
- virtual bool **open** (const QString & db, const QString & user = QString::null, const QString & password = QString::null, const QString & host = QString::null, int port = -1)
- virtual void **close** ()
- virtual QSqlQuery **createQuery** () const

Protected Members

- virtual void **setOpen** (bool o)
- virtual void **setOpenError** (bool e)
- virtual void **setLastError** (const QSqlError & e)

Detailed Description

The QSqlDriver class is an abstract base class for accessing SQL databases.

This class should not be used directly. Use QSqlDatabase instead.

See also Database Classes.

Member Type Documentation

QSqlDriver::DriverFeature

This enum contains a list of features a driver may support. Use hasFeature() to query whether a feature is supported or not.

The currently defined values are:

- `QSqlDriver::Transactions` - whether the driver supports SQL transactions
- `QSqlDriver::QuerySize` - whether the database is capable of reporting the size of a query. Note that some databases do not support returning the size (i.e. number of rows returned) of a query, in which case QSqlQuery::size() will return -1
- `QSqlDriver::BLOB` - whether the driver supports Binary Large Object fields

See also hasFeature() [p. 89].

Member Function Documentation

QSqlDriver::QSqlDriver (QObject * parent = 0, const char * name = 0)

Default constructor. Creates a new driver with parent *parent* and name *name*.

QSqlDriver::~QSqlDriver ()

Destroys the object and frees any allocated resources.

bool QSqlDriver::beginTransaction () [virtual]

Protected function which derived classes can reimplement to begin a transaction. If successful, return TRUE, otherwise return FALSE. The default implementation returns FALSE.

See also commitTransaction() [p. 89] and rollbackTransaction() [p. 90].

void QSqlDriver::close () [virtual]

Derived classes must reimplement this abstract virtual function in order to close the database connection. Return TRUE on success, FALSE on failure.

See also setOpen() [p. 91].

bool QSqlDriver::commitTransaction () [virtual]

Protected function which derived classes can reimplement to commit a transaction. If successful, return TRUE, otherwise return FALSE. The default implementation returns FALSE.

See also beginTransaction() [p. 88] and rollbackTransaction() [p. 90].

QSqlQuery QSqlDriver::createQuery () const [virtual]

Creates an empty SQL result on the database. Derived classes must reimplement this function and return a QSqlQuery object appropriate for their database to the caller.

QString QSqlDriver::formatValue (const QSqlField * field, bool trimStrings = FALSE) const [virtual]

Returns a string representation of the *field* value for the database. This is used, for example, when constructing INSERT and UPDATE statements.

The default implementation returns the value formatted as a string according to the following rules:

- If *field* is null, nullText() is returned.
- If *field* is character data, the value is returned enclosed in single quotation marks, which is appropriate for many SQL databases. Any embedded single-quote characters are escaped (replaced with two single-quote characters). If *trimStrings* is TRUE (the default is FALSE), all trailing whitespace is trimmed from the field.
- If *field* is date/time data, the value is formatted in ISO format and enclosed in single quotation marks. If the date/time data is invalid, nullText() is returned.
- If *field* is bytearray data, and the driver can edit binary fields, the value is formatted as a hexadecimal string.
- For any other field type toString() will be called on its value and the result returned.

See also QVariant::toString() [Datastructures and String Handling with Qt].

bool QSqlDriver::hasFeature (DriverFeature f) const [virtual]

Returns TRUE if the driver supports feature *f*; otherwise returns FALSE.

Note that some databases need to be open() before this can be determined.

See also DriverFeature [p. 88].

bool QSqlDriver::isOpen () const

Returns TRUE if the database connection is open, FALSE otherwise.

bool QSqlDriver::isOpenError () const

Returns TRUE if the there was an error opening the database connection, FALSE otherwise.

QSqlError QSqlDriver::lastError () const

Returns a QSqlError object which contains information about the last error that occurred on the database.

QString QSqlDriver::nullText () const [virtual]

Returns a string representation of the 'NULL' value for the database. This is used, for example, when constructing INSERT and UPDATE statements. The default implementation returns the string 'NULL'.

bool QSqlDriver::open (const QString & db, const QString & user = QString::null, const QString & password = QString::null, const QString & host = QString::null, int port = -1) [virtual]

Derived classes must reimplement this abstract virtual function in order to open a database connection on database *db*, using user name *user*, password *password*, host *host* and port *port*.

The function *must* return TRUE on success and FALSE on failure.

See also setOpen() [p. 91].

QSqlIndex QSqlDriver::primaryIndex (const QString & tableName) const [virtual]

Returns the primary index for table *tableName*. Returns an empty QSqlIndex if the table doesn't have a primary index. The default implementation returns an empty index.

QSqlRecord QSqlDriver::record (const QString & tableName) const [virtual]

Returns a QSqlRecord populated with the names of the fields in table *tableName*. If no such table exists, an empty list is returned. The default implementation returns an empty record.

QSqlRecord QSqlDriver::record (const QSqlQuery & query) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a QSqlRecord populated with the names of the fields in the SQL *query*. The default implementation returns an empty record.

QSqlRecordInfo QSqlDriver::recordInfo (const QString & tablename) const [virtual]

Returns a QSqlRecordInfo object with meta data on the table *tablename*.

QSqlRecordInfo QSqlDriver::recordInfo (const QSqlQuery & query) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a QSqlRecordInfo object with meta data for the QSqlQuery *query*. Note that this overloaded function may return not as much information as the recordInfo function which takes the name of a table as parameter.

bool QSqlDriver::rollbackTransaction () [virtual]

Protected function which derived classes can reimplement to rollback a transaction. If successful, return TRUE, otherwise return FALSE. The default implementation returns FALSE.

See also beginTransaction() [p. 88] and commitTransaction() [p. 89].

void QSqlDriver::setLastError (const QSqlError & e) [virtual protected]

Protected function which allows derived classes to set the value of the last error, *e*, that occurred on the database.

See also lastError() [p. 89].

void QSqlDriver::setOpen (bool o) [virtual protected]

Protected function which sets the open state of the database to *o*. Derived classes can use this function to report the status of open().

See also open() [p. 90] and setOpenError() [p. 91].

void QSqlDriver::setOpenError (bool e) [virtual protected]

Protected function which sets the open error state of the database to *e*. Derived classes can use this function to report the status of open(). Note that if *e* is TRUE the open state of the database is set to closed (i.e., isOpen() returns FALSE).

See also open() [p. 90].

QStringList QSqlDriver::tables (const QString & user) const [virtual]

Returns a list of tables in the database. The default implementation returns an empty list.

Currently the *user* argument is unused.

QSqlEditorFactory Class Reference

The QSqlEditorFactory class is used to create the editors used by QDataTable and QSqlForm.

This class is part of the **sql** module.

`#include <qsqleditorfactory.h>`

Inherits QEditorFactory [Additional Functionality with Qt].

Public Members

- **QSqlEditorFactory** (QObject * parent = 0, const char * name = 0)
- **~QSqlEditorFactory** ()
- virtual QWidget * **createEditor** (QWidget * parent, const QVariant & variant)
- virtual QWidget * **createEditor** (QWidget * parent, const QSqlField * field)

Static Public Members

- QSqlEditorFactory * **defaultFactory** ()
- void **installDefaultFactory** (QSqlEditorFactory * factory)

Detailed Description

The QSqlEditorFactory class is used to create the editors used by QDataTable and QSqlForm.

QSqlEditorFactory is used by QDataTable and QSqlForm to automatically create appropriate editors for a given QSqlField. For example if the field is a QVariant::String a QLineEdit would be the default editor, whereas a QVariant::Int's default editor would be a QSpinBox.

If you want to create different editors for fields with the same data type, subclass QSqlEditorFactory and reimplement the createEditor() function.

See also QDataTable [p. 47], QSqlForm [p. 105] and Database Classes.

Member Function Documentation

QSqlEditorFactory::QSqlEditorFactory (QObject * parent = 0, const char * name = 0)

Constructs a SQL editor factory with parent *parent* and name *name*.

QSqlEditorFactory::~QSqlEditorFactory ()

Destroys the object and frees any allocated resources.

QWidget * QSqlEditorFactory::createEditor (QWidget * parent, const QVariant & variant) [virtual]

Creates and returns the appropriate editor widget for the QVariant *variant*.

The widget that is returned has the parent *parent* (which may be zero). If *variant* is invalid, 0 is returned.

Reimplemented from QEditorFactory [Additional Functionality with Qt].

QWidget * QSqlEditorFactory::createEditor (QWidget * parent, const QSqlField * field) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Creates and returns the appropriate editor for the QSqlField *field*.

QSqlEditorFactory * QSqlEditorFactory::defaultFactory () [static]

Returns an instance of a default editor factory.

void QSqlEditorFactory::installDefaultFactory (QSqlEditorFactory * factory) [static]

Replaces the default editor factory with *factory*. All QDataTable and QSqlForm instantiations will use this new factory for creating field editors. *QSqlEditorFactory takes ownership of factory, and destroys it when it is no longer needed.*

QSqlError Class Reference

The QSqlError class provides SQL database error information.

This class is part of the **sql** module.

```
#include <qsqlerror.h>
```

Public Members

- enum **Type** { None, Connection, Statement, Transaction, Unknown }
- **QSqlError** (const QString & driverText = QString::null, const QString & databaseText = QString::null, int type = QSqlError::None, int number = -1)
- **QSqlError** (const QSqlError & other)
- QSqlError & **operator=** (const QSqlError & other)
- virtual **~QSqlError** ()
- QString **driverText** () const
- virtual void **setDriverText** (const QString & driverText)
- QString **databaseText** () const
- virtual void **setDatabaseText** (const QString & databaseText)
- int **type** () const
- virtual void **setType** (int type)
- int **number** () const
- virtual void **setNumber** (int number)

Detailed Description

The QSqlError class provides SQL database error information.

This class is used to report database-specific errors. An error description and (if appropriate) a database-specific error number can be recovered using this class.

See also Database Classes.

Member Type Documentation

QSqlError::Type

This enum type describes the type of SQL error that occurred.

The currently defined values are:

- `QSqlError::None` - no error occurred
- `QSqlError::Connection` - connection error
- `QSqlError::Statement` - statement syntax error
- `QSqlError::Transaction` - transaction failed error
- `QSqlError::Unknown` - unknown error

Member Function Documentation

QSqlError::QSqlError (const QString & driverText = QString::null, const QString & databaseText = QString::null, int type = QSqlError::None, int number = -1)

Constructs an error containing the driver error text *driverText*, the database-specific error text *databaseText*, the type *type* and the optional error number *number*.

QSqlError::QSqlError (const QSqlError & other)

Creates a copy of *other*.

QSqlError::~QSqlError () [virtual]

Destroys the object and frees any allocated resources.

QString QSqlError::databaseText () const

Returns the text of the error as reported by the database. This may contain database-specific descriptions.

QString QSqlError::driverText () const

Returns the text of the error as reported by the driver. This may contain database-specific descriptions.

int QSqlError::number () const

Returns the database-specific error number, or -1 if it cannot be determined.

QSqlError & QSqlError::operator= (const QSqlError & other)

Sets the error equal to *other*.

void QSqlError::setDatabaseText (const QString & databaseText) [virtual]

Sets the database error text to the value of *databaseText*.

void QSqlError::setDriverText (const QString & driverText) [virtual]

Sets the driver error text to the value of *driverText*.

void QSqlError::setNumber (int number) [virtual]

Sets the database-specific error number to the value of *number*.

void QSqlError::setType (int type) [virtual]

Sets the error type to the value of *type*.

int QSqlError::type () const

Returns the error type, or -1 if the type cannot be determined.

See also QSqlError::Type [p. 94].

QSqlField Class Reference

The QSqlField class manipulates the fields in SQL database tables and views.

This class is part of the **sql** module.

`#include <qsqlfield.h>`

Public Members

- **QSqlField** (const QString & fieldName = QString::null, QVariant::Type type = QVariant::Invalid)
- **QSqlField** (const QSqlField & other)
- QSqlField & **operator=** (const QSqlField & other)
- bool **operator==** (const QSqlField & other) const
- virtual **~QSqlField** ()
- virtual void **setValue** (const QVariant & value)
- virtual QVariant **value** () const
- virtual void **setName** (const QString & name)
- QString **name** () const
- virtual void **setNull** ()
- bool **isNull** () const
- virtual void **setReadOnly** (bool readOnly)
- bool **isReadOnly** () const
- void **clear** (bool nullify = TRUE)
- QVariant::Type **type** () const

Detailed Description

The QSqlField class manipulates the fields in SQL database tables and views.

QSqlField represents the characteristics of a single column in a database table or view, such as the data type and column name. A field also contains the value of the database column, which can be viewed or changed.

Field data values are stored as QVariants. Using an incompatible type is not permitted. For example:

```
QSqlField f( "myfield", QVariant::Int );
f.setValue( QPixmap() );  // will not work
```

However, the field will attempt to cast certain data types to the field data type where possible:

```
QSqlField f( "myfield", QVariant::Int );
f.setValue( QString("123") ); // casts QString to int
```

QSqlField objects are rarely created explicitly in application code. They are usually accessed indirectly through QSqlRecord or QSqlCursor which already contain a list of fields. For example:

```
QSqlCursor cur( "Employee" );        // create cursor using the 'Employee' table
QSqlField* f = cur.field( "name" );  // use the 'name' field
f->setValue( "Dave" );               // set field value
...
```

In practice we rarely need to extract a pointer to a field at all. The previous example would normally be written:

```
QSqlCursor cur( "Employee" );
cur.setValue( "name", "Dave" );
...
```

See also Database Classes.

Member Function Documentation

QSqlField::QSqlField (const QString & fieldName = QString::null, QVariant::Type type = QVariant::Invalid)

Constructs an empty field called *fieldName* of type *type*.

QSqlField::QSqlField (const QSqlField & other)

Constructs a copy of *other*.

QSqlField::~QSqlField () [virtual]

Destroys the object and frees any allocated resources.

void QSqlField::clear (bool nullify = TRUE)

Clears the value of the field. If the field is read-only, nothing happens. If *nullify* is TRUE (the default), the field is set to NULL.

bool QSqlField::isNull () const

Returns TRUE if the field is currently null, otherwise returns FALSE.

bool QSqlField::isReadOnly () const

Returns TRUE if the field's value is read only, otherwise FALSE.

QString QSqlField::name () const

Returns the name of the field.

Example: sql/overview/table4/main.cpp.

QSqlField & QSqlField::operator= (const QSqlField & other)

Sets the field equal to *other*.

bool QSqlField::operator== (const QSqlField & other) const

Returns TRUE if the field is equal to *other*, otherwise returns FALSE. Fields are considered equal when the following field properties are the same:

- name()
- isNull()
- value()
- isReadOnly()

void QSqlField::setName (const QString & name) [virtual]

Sets the name of the field to *name*.

void QSqlField::setNull () [virtual]

Sets the field to NULL and clears the value using clear(). If the field is read-only, nothing happens.

See also isReadOnly() [p. 98] and clear() [p. 98].

void QSqlField::setReadOnly (bool readOnly) [virtual]

Sets the read only flag of the field's value to *readOnly*.

See also setValue() [p. 99].

void QSqlField::setValue (const QVariant & value) [virtual]

Sets the value of the field to *value*. If the field is read-only (isReadOnly() returns TRUE), nothing happens. If the data type of *value* differs from the field's current data type, an attempt is made to cast it to the proper type. This preserves the data type of the field in the case of assignment, e.g. a QString to an integer data type. For example:

```
QSqlCursor cur( "Employee" );                // 'Employee' table
QSqlField* f = cur.field( "student_count" );  // an integer field
...
f->setValue( myLineEdit->text() );           // cast the line edit text to an integer
```

See also isReadOnly() [p. 98].

QVariant::Type QSqlField::type () const

Returns the field's type.

QVariant QSqlField::value () const [virtual]

Returns the internal value of the field as a QVariant.

Example: sql/overview/table4/main.cpp.

QSqlFieldInfo Class Reference

The API for this class is under development and is subject to change.
We do not recommend the use of this class for production work at this time.

The QSqlFieldInfo class stores meta data associated with a SQL field.

This class is part of the **sql** module.

```
#include <qsqlfield.h>
```

Public Members

- **QSqlFieldInfo** (const QString & name = QString::null, QVariant::Type typ = QVariant::Invalid, int required = -1, int len = -1, int prec = -1, const QVariant & defValue = QVariant ( ), int typeID = 0, bool generated = TRUE, bool trim = FALSE, bool calculated = FALSE)
- **QSqlFieldInfo** (const QSqlFieldInfo & other)
- **QSqlFieldInfo** (const QSqlField & other, bool generated = TRUE)
- virtual **~QSqlFieldInfo** ()
- QSqlFieldInfo & **operator=** (const QSqlFieldInfo & other)
- bool **operator==** (const QSqlFieldInfo & f) const
- QSqlField **toField** () const
- int **isRequired** () const
- QVariant::Type **type** () const
- int **length** () const
- int **precision** () const
- QVariant **defaultValue** () const
- QString **name** () const
- int **typeID** () const
- bool **isGenerated** () const
- bool **isTrim** () const
- bool **isCalculated** () const
- virtual void **setTrim** (bool trim)
- virtual void **setGenerated** (bool gen)
- virtual void **setCalculated** (bool calc)

Detailed Description

The QSqlFieldInfo class stores meta data associated with a SQL field.

QSqlFieldInfo objects only store meta data; field values are stored in QSqlField objects.

All values must be set in the constructor, and may be retrieved using isRequired(), type(), length(), precision(), defaultValue(), name(), isGenerated() and typeID().

See also Database Classes.

Member Function Documentation

QSqlFieldInfo::QSqlFieldInfo (const QString & name = QString::null, QVariant::Type typ = QVariant::Invalid, int required = -1, int len = -1, int prec = -1, const QVariant & defValue = QVariant ( ), int typeID = 0, bool generated = TRUE, bool trim = FALSE, bool calculated = FALSE)

Constructs a QSqlFieldInfo with the following parameters:

- *name* the name of the field.
- *typ* the field's type in a QVariant.
- *required* greater than 0 if the field is required, 0 if its value can be NULL and less than 0 if it cannot be determined whether the field is required or not.
- *len* the length of the field. Note that for non-character types some databases return either the length in bytes or the number of digits. -1 signifies that the length cannot be determined.
- *prec* the precision of the field, or -1 if the field has no precision or it cannot be determined.
- *defValue* the default value that is inserted into the table if none is specified by the user. QVariant() if there is no default value or it cannot be determined.
- *typeID* the internal typeID of the database system (only useful for low-level programming). 0 if unknown.
- *generated* TRUE indicates that this field should be included in auto-generated SQL statments, e.g. in QSqlCursor.
- *trim* TRUE indicates that widgets should remove trailing whitespace from character fields. This does not affect the field value but only its representation inside widgets.
- *calculated* TRUE indicates that the value of this field is calculated. The value of calculated fields can by modified by subclassing QSqlCursor and overriding QSqlCursor::calculateField().

QSqlFieldInfo::QSqlFieldInfo (const QSqlFieldInfo & other)

Constructs a copy of *other*.

QSqlFieldInfo::QSqlFieldInfo (const QSqlField & other, bool generated = TRUE)

Creates a QSqlFieldInfo object with the type and the name of the QSqlField *other*. If *generated* is TRUE this field will be included in auto-generated SQL statments, e.g. in QSqlCursor.

QSqlFieldInfo::~QSqlFieldInfo () [virtual]

Destroys the object and frees any allocated resources.

QVariant QSqlFieldInfo::defaultValue () const

Returns the default value of this field or an empty QVariant if the field has no default value or the value couldn't be determined. The default value is the value inserted in the database when it was not explicitly specified by the user.

bool QSqlFieldInfo::isCalculated () const

Returns TRUE if the field is calculated.

See also setCalculated() [p. 104].

bool QSqlFieldInfo::isGenerated () const

Returns TRUE if this field should be included in auto-generated SQL statments, e.g. in QSqlCursor; otherwise returns FALSE.

See also setGenerated() [p. 104].

int QSqlFieldInfo::isRequired () const

Returns a value greater than 0 if the field is required (NULL values are not allowed), 0 if it isn't required (NULL values are allowed) or less than 0 if it cannot be determined whether the field is required or not.

bool QSqlFieldInfo::isTrim () const

Returns TRUE if trailing whitespace should be removed from character fields.

See also setTrim() [p. 104].

int QSqlFieldInfo::length () const

Returns the length of this field. For fields storing text the return value is the maximum number of characters the field can hold. For non-character fields some database systems return the number of bytes needed or the number of digits allowed. If the length cannot be determined -1 is returned.

QString QSqlFieldInfo::name () const

Returns the name of the field in the SQL table.

Examples: sql/overview/subclass3/main.cpp and sql/overview/subclass4/main.cpp.

QSqlFieldInfo & QSqlFieldInfo::operator= (const QSqlFieldInfo & other)

Assigns *other* to this field info and returns a reference to it.

bool QSqlFieldInfo::operator== (const QSqlFieldInfo & f) const

Returns TRUE if this fieldinfo is equal to *f*; otherwise returns FALSE.

Two field infos are considered equal when all their attributes match.

int QSqlFieldInfo::precision () const

Returns the precision of this field or -1 if the field has no precision or it cannot be determined.

void QSqlFieldInfo::setCalculated (bool calc) [virtual]

calc set to TRUE indicates that this field is a calculated field. The value of calculated fields can by modified by subclassing QSqlCursor and overriding QSqlCursor::calculateField().

See also isCalculated() [p. 103].

void QSqlFieldInfo::setGenerated (bool gen) [virtual]

gen set to FALSE indicates that this field should not appear in auto-generated SQL statements (for example in QSqlCursor).

See also isGenerated() [p. 103].

void QSqlFieldInfo::setTrim (bool trim) [virtual]

If *trim* is TRUE widgets should remove trailing whitespace from character fields. This does not affect the field value but only its representation inside widgets.

See also isTrim() [p. 103].

QSqlField QSqlFieldInfo::toField () const

Returns an empty QSqlField based on the information in this QSqlFieldInfo.

QVariant::Type QSqlFieldInfo::type () const

Returns the type of this field or QVariant::Invalid if the type is unknown.

int QSqlFieldInfo::typeID () const

Returns the internal type identifier as returned from the database system. The return value is 0 if the type is unknown.

Warning: This information is only useful for low-level database programming and is *not* database independent.

QSqlForm Class Reference

The QSqlForm class creates and manages data entry forms tied to SQL databases.

This class is part of the **sql** module.

`#include <qsqlform.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QSqlForm** (QObject * parent = 0, const char * name = 0)
- **~QSqlForm** ()
- virtual void **insert** (QWidget * widget, const QString & field)
- virtual void **remove** (const QString & field)
- uint **count** () const
- QWidget * **widget** (uint i) const
- QSqlField * **widgetToField** (QWidget * widget) const
- QWidget * **fieldToWidget** (QSqlField * field) const
- void **installPropertyMap** (QSqlPropertyMap * pmap)
- virtual void **setRecord** (QSqlRecord * buf)

Public Slots

- virtual void **readField** (QWidget * widget)
- virtual void **writeField** (QWidget * widget)
- virtual void **readFields** ()
- virtual void **writeFields** ()
- virtual void **clear** ()
- virtual void **clearValues** (bool nullify = FALSE)

Protected Members

- virtual void **insert** (QWidget * widget, QSqlField * field)
- virtual void **remove** (QWidget * widget)

Detailed Description

The QSqlForm class creates and manages data entry forms tied to SQL databases.

Typical use of a QSqlForm consists of the following steps:

1. Create the widgets you want to appear in the form.
2. Create a cursor and navigate to the record to be edited.
3. Create the QSqlForm.
4. Set the form's record buffer to the cursor's update buffer.
5. Insert each widget and the field it is to edit into the form.
6. Use readFields() to update the editor widgets with values from the database's fields.
7. Display the form and let the user edit values etc.
8. Use writeFields() to update the database's field values with the values in the editor widgets.

Note that a QSqlForm does not access the database directly, but most often via QSqlFields which are part of a QSqlCursor. A QSqlCursor::insert(), QSqlCursor::update() or QSqlCursor::del() call is needed to actually write values to the database.

Some sample code to initialize a form successfully:

```
QLineEdit  myEditor( this );
QSqlForm   myForm( this );
QSqlCursor myCursor( "mytable" );

// Execute a query to make the cursor valid
myCursor.select();
// Move the cursor to a valid record (the first record)
myCursor.next();
// Set the form's record pointer to the cursor's edit buffer (which
// contains the current record's values)
myForm.setRecord( myCursor.primeUpdate() );

// Insert a field into the form that uses myEditor to edit the
// field 'somefield' in 'mytable'
myForm.insert( &myEditor, "somefield" );

// Update myEditor with the value from the mapped database field
myForm.readFields();
...
// Let the user edit the form
...
// Update the database
myForm.writeFields(); // Update the cursor's edit buffer from the form
myCursor.update();    // Update the database from the cursor's buffer
```

If you want to use custom editors for displaying/editing data fields, you need to install a custom QSqlPropertyMap. The form uses this object to get or set the value of a widget.

Note that *Qt Designer* provides a visual means of creating data-aware forms.

See also installPropertyMap() [p. 107], QSqlPropertyMap [p. 112] and Database Classes.

Member Function Documentation

QSqlForm::QSqlForm (QObject * parent = 0, const char * name = 0)

Constructs a QSqlForm with parent *parent* and name *name*.

QSqlForm::~QSqlForm ()

Destroys the object and frees any allocated resources.

void QSqlForm::clear () [virtual slot]

Removes every widget, and the fields they're mapped to, from the form.

void QSqlForm::clearValues (bool nullify = FALSE) [virtual slot]

Clears the values in all the widgets, and the fields they are mapped to, in the form. If *nullify* is TRUE (the default is FALSE), each field is also set to null.

uint QSqlForm::count () const

Returns the number of widgets in the form.

QWidget * QSqlForm::fieldToWidget (QSqlField * field) const

Returns the widget that field *field* is mapped to.

void QSqlForm::insert (QWidget * widget, const QString & field) [virtual]

Inserts a *widget*, and the name of the *field* it is to be mapped to, into the form. To actually associate inserted widgets with an edit buffer, use setRecord().

See also setRecord() [p. 108].

Examples: sql/overview/form1/main.cpp and sql/overview/form2/main.cpp.

void QSqlForm::insert (QWidget * widget, QSqlField * field) [virtual protected]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Inserts a *widget*, and the *field* it is to be mapped to, into the form.

void QSqlForm::installPropertyMap (QSqlPropertyMap * pmap)

Installs a custom QSqlPropertyMap. This is useful if you plan to create your own custom editor widgets.

QSqlForm takes ownership of *pmap*, and *pmap* is therefore deleted when QSqlForm goes out of scope.

See also QDataTable::installEditorFactory() [p. 54].

Example: sql/overview/custom1/main.cpp.

void QSqlForm::readField (QWidget * widget) [virtual slot]

Updates the widget *widget* with the value from the SQL field it is mapped to. Nothing happens if no SQL field is mapped to the *widget*.

void QSqlForm::readFields () [virtual slot]

Updates the widgets in the form with current values from the SQL fields they are mapped to.

Examples: sql/overview/form1/main.cpp and sql/overview/form2/main.cpp.

void QSqlForm::remove (QWidget * widget) [virtual protected]

Removes a *widget*, and hence the field it's mapped to, from the form.

void QSqlForm::remove (const QString & field) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Removes *field* from the form.

void QSqlForm::setRecord (QSqlRecord * buf) [virtual]

Sets *buf* as the record buffer for the form. To force the display of the data from *buf*, use readFields().

See also readFields() [p. 108] and writeFields() [p. 108].

Examples: sql/overview/custom1/main.cpp, sql/overview/form1/main.cpp and sql/overview/form2/main.cpp.

QWidget * QSqlForm::widget (uint i) const

Returns the *i*-th widget in the form. Useful for traversing the widgets in the form.

QSqlField * QSqlForm::widgetToField (QWidget * widget) const

Returns the SQL field that widget *widget* is mapped to.

void QSqlForm::writeField (QWidget * widget) [virtual slot]

Updates the SQL field with the value from the *widget* it is mapped to. Nothing happens if no SQL field is mapped to the *widget*.

void QSqlForm::writeFields () [virtual slot]

Updates the SQL fields with values from the widgets they are mapped to. To actually update the database with the contents of the record buffer, use QSqlCursor::insert(), QSqlCursor::update() or QSqlCursor::del() as appropriate.

Example: sql/overview/form2/main.cpp.

QSqlIndex Class Reference

The QSqlIndex class provides functions to manipulate and describe QSqlCursor and QSqlDatabase indexes.

This class is part of the **sql** module.

`#include <qsqlindex.h>`

Inherits QSqlRecord [p. 122].

Public Members

- **QSqlIndex** (const QString & cursorname = QString::null, const QString & name = QString::null)
- **QSqlIndex** (const QSqlIndex & other)
- **~QSqlIndex** ()
- QSqlIndex & **operator=** (const QSqlIndex & other)
- virtual void **setCursorName** (const QString & cursorName)
- QString **cursorName** () const
- virtual void **setName** (const QString & name)
- QString **name** () const
- virtual void **append** (const QSqlField & field)
- virtual void **append** (const QSqlField & field, bool desc)
- bool **isDescending** (int i) const
- virtual void **setDescending** (int i, bool desc)

Static Public Members

- QSqlIndex **fromStringList** (const QStringList & l, const QSqlCursor * cursor)

Detailed Description

The QSqlIndex class provides functions to manipulate and describe QSqlCursor and QSqlDatabase indexes.

This class is used to describe and manipulate QSqlCursor and QSqlDatabase indexes. An index refers to a single table or view in a database. Information about the fields that comprise the index can be used to generate SQL statements, or to affect the behavior of a QSqlCursor object.

Normally, QSqlIndex objects are created by QSqlDatabase or QSqlCursor.

See also Database Classes.

Member Function Documentation

QSqlIndex::QSqlIndex (const QString & cursorname = QString::null, const QString & name = QString::null)

Constructs an empty index using the cursor name *cursorname* and index name *name*.

QSqlIndex::QSqlIndex (const QSqlIndex & other)

Constructs a copy of *other*.

QSqlIndex::~QSqlIndex ()

Destroys the object and frees any allocated resources.

void QSqlIndex::append (const QSqlField & field) [virtual]

Appends the field *field* to the list of indexed fields. The field is appended with an ascending sort order.

Reimplemented from QSqlRecord [p. 123].

void QSqlIndex::append (const QSqlField & field, bool desc) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Appends the field *field* to the list of indexed fields. The field is appended with an ascending sort order, unless *desc* is TRUE.

QString QSqlIndex::cursorName () const

Returns the name of the cursor which the index is associated with.

QSqlIndex QSqlIndex::fromStringList (const QStringList & l, const QSqlCursor * cursor) [static]

Returns an index based on the field descriptions in *l* and the cursor *cursor*. The field descriptions should be in the same format that toStringList() produces, for example, a surname field in the people table might be in one of these forms: "surname", "surname DESC" or "people.surname ASC".

See also toStringList() [p. 126].

bool QSqlIndex::isDescending (int i) const

Returns true if field *i* in the index is sorted in descending order, otherwise returns FALSE.

QString QSqlIndex::name () const

Returns the name of the index.

QSqlIndex & QSqlIndex::operator= (const QSqlIndex & other)

Sets the index equal to *other*.

void QSqlIndex::setCursorName (const QString & cursorName) [virtual]

Sets the name of the cursor that the index is associated with to *cursorName*.

void QSqlIndex::setDescending (int i, bool desc) [virtual]

If *desc* is TRUE, field *i* is sorted in descending order. Otherwise, field *i* is sorted in ascending order (the default). If the field does not exist, nothing happens.

void QSqlIndex::setName (const QString & name) [virtual]

Sets the name of the index to *name*.

QSqlPropertyMap Class Reference

The QSqlPropertyMap class is used to map widgets to SQL fields.

This class is part of the **sql** module.

`#include <qsqlpropertymap.h>`

Public Members

- **QSqlPropertyMap** ()
- virtual **~QSqlPropertyMap** ()
- QVariant **property** (QWidget * widget)
- virtual void **setProperty** (QWidget * widget, const QVariant & value)
- void **insert** (const QString & classname, const QString & property)
- void **remove** (const QString & classname)

Static Public Members

- QSqlPropertyMap * **defaultMap** ()
- void **installDefaultMap** (QSqlPropertyMap * map)

Detailed Description

The QSqlPropertyMap class is used to map widgets to SQL fields.

The SQL module uses Qt object properties to insert and extract values from editor widgets.

This class is used to map editors to SQL fields. This works by associating SQL editor class names to the properties used to insert and extract values to/from the editor.

For example, a QLineEdit can be used to edit text strings and other data types in QDataTables or QSqlForms. Several properties are defined in QLineEdit, but only the *text* property is used to insert and extract text from a QLineEdit. Both QDataTable and QSqlForm use the global QSqlPropertyMap for inserting and extracting values to and from an editor widget. The global property map defines several common widgets and properties that are suitable for many applications. You can add and remove widget properties to suit your specific needs.

If you want to use custom editors with your QDataTable or QSqlForm, you have to install your own QSqlPropertyMap for that table or form. Example:

```
QSqlPropertyMap *myMap  = new QSqlPropertyMap();
QSqlForm        *myForm = new QSqlForm( this );
MyEditor         myEditor( this );
```

```
// Set the QSqlForm's record buffer to the update buffer of
// a pre-existing QSqlCursor called 'cur'.
myForm->setRecord( cur->primeUpdate() );

// Install the customized map
myMap->insert( "MyEditor", "content" );
myForm->installPropertyMap( myMap ); // myForm now owns myMap
...
// Insert a field into the form that uses a myEditor to edit the
// field 'somefield'
myForm->insert( &myEditor, "somefield" );

// Update myEditor with the value from the mapped database field
myForm->readFields();
...
// Let the user edit the form
...
// Update the database fields with the values in the form
myForm->writeFields();
...
```

You can also replace the global QSqlPropertyMap that is used by default. (Bear in mind that QSqlPropertyMap takes ownership of the new default map.)

```
QSqlPropertyMap *myMap = new QSqlPropertyMap;

myMap->insert( "MyEditor", "content" );
QSqlPropertyMap::installDefaultMap( myMap );
...
```

See also QDataTable [p. 47], QSqlForm [p. 105], QSqlEditorFactory [p. 92] and Database Classes.

Member Function Documentation

QSqlPropertyMap::QSqlPropertyMap ()

Constructs a QSqlPropertyMap.

The default property mappings used by Qt widgets are:

- QButton — text
- QCheckBox — checked
- QComboBox — currentItem
- QDateEdit — date
- QDateTimeEdit — dateTime
- QDial — value
- QLabel — text
- QLCDNumber — value
- QLineEdit — text
- QListBox — currentItem
- QMultiLineEdit — text

- QPushButton — text
- QRadioButton — text
- QScrollBar — value
- QSlider — value
- QSpinBox — value
- QTextBrowser — source
- QTextEdit — text
- QTextView — text
- QTimeEdit — time

QSqlPropertyMap::~QSqlPropertyMap () [virtual]

Destroys the QSqlPropertyMap.

Note that if the QSqlPropertyMap is installed with installPropertyMap() the object it was installed into, e.g. the QSqlForm, takes ownership and will delete the QSqlPropertyMap when necessary.

QSqlPropertyMap * QSqlPropertyMap::defaultMap () [static]

Returns the application global QSqlPropertyMap.

void QSqlPropertyMap::insert (const QString & classname, const QString & property)

Insert a new classname/property pair, which is used for custom SQL field editors. There *must* be a Q_PROPERTY clause in the *classname* class declaration for the *property*.

Example: sql/overview/custom1/main.cpp.

void QSqlPropertyMap::installDefaultMap (QSqlPropertyMap * map) [static]

Replaces the global default property map with *map*. All QDataTable and QSqlForm instantiations will use this new map for inserting and extracting values to and from editors. *QSqlPropertyMap takes ownership of map*, and destroys it when it is no longer needed.

QVariant QSqlPropertyMap::property (QWidget * widget)

Returns the mapped property of *widget* as a QVariant.

void QSqlPropertyMap::remove (const QString & classname)

Removes *classname* from the map.

void QSqlPropertyMap::setProperty (QWidget * widget, const QVariant & value) [virtual]

Sets the property of *widget* to *value*.

QSqlQuery Class Reference

The QSqlQuery class provides a means of executing and manipulating SQL statements.

This class is part of the **sql** module.

`#include <qsqlquery.h>`

Inherited by QSqlCursor [p. 67].

Public Members

- **QSqlQuery** (QSqlResult * r)
- **QSqlQuery** (const QString & query = QString::null, QSqlDatabase * db = 0)
- **QSqlQuery** (const QSqlQuery & other)
- QSqlQuery & **operator=** (const QSqlQuery & other)
- virtual **~QSqlQuery** ()
- bool **isValid** () const
- bool **isActive** () const
- bool **isNull** (int field) const
- int **at** () const
- QString **lastQuery** () const
- int **numRowsAffected** () const
- QSqlError **lastError** () const
- bool **isSelect** () const
- int **size** () const
- const QSqlDriver * **driver** () const
- const QSqlResult * **result** () const
- virtual bool **exec** (const QString & query)
- virtual QVariant **value** (int i) const
- virtual bool **seek** (int i, bool relative = FALSE)
- virtual bool **next** ()
- virtual bool **prev** ()
- virtual bool **first** ()
- virtual bool **last** ()

Protected Members

- virtual void **beforeSeek** ()
- virtual void **afterSeek** ()

Detailed Description

The QSqlQuery class provides a means of executing and manipulating SQL statements.

QSqlQuery encapsulates the functionality involved in creating, navigating and retrieving data from SQL queries which are executed on a QSqlDatabase. It can be used to execute DML (data manipulation language) statements, e.g. `SELECT`, `INSERT`, `UPDATE` and `DELETE`, and also DDL (data definition language) statements, e.g. `CREATE TABLE`. It can also be used to execute database-specific commands which are not standard SQL (e.g. `SET DATESTYLE=ISO` for PostgreSQL).

Successfully executed SQL statements set the query to an active state (isActive() returns TRUE) otherwise the query is set to an inactive state. In either case, when executing a new SQL statement, the query is positioned on an invalid record; an active query must be navigated to a valid record (so that isValid() returns TRUE) before values can be retrieved.

Navigating records is performed with the following functions:

- next()
- prev()
- first()
- last()
- seek(int)

These functions allow the programmer to move forward, backward or arbitrarily through the records returned by the query. Once an active query is positioned on a valid record, data can be retrieved using value(). All data is transferred from the SQL backend using QVariants.

For example:

```
QSqlQuery query( "select name from customer;" );
while ( query.next() ) {
    QString name = query.value(0).toString();
    doSomething( name );
}
```

To access the data returned by a query, use the value() method. Each field in the data returned by a SELECT statement is accessed by passing the index number of the desired field, starting with 0. There are no methods to access a field by name to make sure the usage of QSqlQuery is as optimal as possible (see QSqlCursor for a more flexible interface for selecting data from a table or view in the database).

See also QSqlDatabase [p. 80], QSqlCursor [p. 67], QVariant [Datastructures and String Handling with Qt] and Database Classes.

Member Function Documentation

QSqlQuery::QSqlQuery (QSqlResult * r)

Creates a QSqlQuery object which uses the QSqlResult *r* to communicate with a database.

QSqlQuery::QSqlQuery (const QString & query = QString::null, QSqlDatabase * db = 0)

Creates a QSqlQuery object using the SQL *query* and the database *db*. If *db* is 0, (the default), the application's default database is used.

See also QSqlDatabase [p. 80].

QSqlQuery::QSqlQuery (const QSqlQuery & other)

Constructs a copy of *other*.

QSqlQuery::~QSqlQuery () [virtual]

Destroys the object and frees any allocated resources.

void QSqlQuery::afterSeek () [virtual protected]

Protected virtual function called after the internal record pointer is moved to a new record. The default implementation does nothing.

int QSqlQuery::at () const

Returns the current internal position of the query. The first record is at position zero. If the position is invalid, a QSql::Location will be returned indicating the invalid position.

See also isValid() [p. 118].

Example: sql/overview/navigating/main.cpp.

void QSqlQuery::beforeSeek () [virtual protected]

Protected virtual function called before the internal record pointer is moved to a new record. The default implementation does nothing.

const QSqlDriver * QSqlQuery::driver () const

Returns a pointer to the database driver associated with the query.

bool QSqlQuery::exec (const QString & query) [virtual]

Executes the SQL *query*. Returns TRUE if the query was successful sets the query state to active, otherwise returns FALSE and the query becomes inactive. The *query* string must use syntax appropriate for the SQL database being queried, for example, standard SQL.

After the query is executed, the query is positioned on an invalid record, and must be navigated to a valid record before data values can be retrieved.

See also isActive() [p. 118], isValid() [p. 118], next() [p. 118], prev() [p. 119], first() [p. 117], last() [p. 118] and seek() [p. 120].

Examples: sql/overview/basicbrowsing/main.cpp, sql/overview/basicbrowsing2/main.cpp and sql/overview/basicdatamanip/main.cpp.

bool QSqlQuery::first () [virtual]

Retrieves the first record in the result, if available, and positions the query on the retrieved record. Note that the result must be in an active state and isSelect() must return TRUE before calling this function or it will do nothing and return FALSE. Returns TRUE if successful. If unsuccessful the query position is set to an invalid position and FALSE is returned.

Example: sql/overview/navigating/main.cpp.

bool QSqlQuery::isActive () const

Returns TRUE if the query is currently active, otherwise returns FALSE.

Examples: sql/overview/basicbrowsing/main.cpp, sql/overview/basicbrowsing2/main.cpp, sql/overview/basicdatamanip/main.cpp, sql/overview/navigating/main.cpp and sql/overview/retrieve1/main.cpp.

bool QSqlQuery::isNull (int field) const

Returns TRUE if *field* is currently NULL, otherwise returns FALSE. The query must be active and positioned on a valid record before calling this function otherwise it returns FALSE. Note that, for some drivers, isNull() will not return accurate information until after an attempt is made to retrieve data.

See also isActive() [p. 118], isValid() [p. 118] and value() [p. 120].

bool QSqlQuery::isSelect () const

Returns TRUE if the current query is a `SELECT` statement, otherwise returns FALSE.

bool QSqlQuery::isValid () const

Returns TRUE if the query is currently positioned on a valid record, otherwise returns FALSE.

bool QSqlQuery::last () [virtual]

Retrieves the last record in the result, if available, and positions the query on the retrieved record. Note that the result must be in an active state and isSelect() must return TRUE before calling this function or it will do nothing and return FALSE. Returns TRUE if successful. If unsuccessful the query position is set to an invalid position and FALSE is returned.

Example: sql/overview/navigating/main.cpp.

QSqlError QSqlQuery::lastError () const

Returns error information about the last error (if any) that occurred.

See also QSqlError [p. 94].

QString QSqlQuery::lastQuery () const

Returns the text of the current query being used, or QString::null if there is no current query text.

bool QSqlQuery::next () [virtual]

Retrieves the next record in the result, if available, and positions the query on the retrieved record. Note that the result must be in an active state and isSelect() must return TRUE before calling this function or it will do nothing and return FALSE.

The following rules apply:

- If the result is currently located before the first record, e.g. immediately after a query is executed, an attempt is made to retrieve the first record.
- If the result is currently located after the last record, there is no change and FALSE is returned.
- If the result is located somewhere in the middle, an attempt is made to retrieve the next record.

If the record could not be retrieved, the result is positioned after the last record and FALSE is returned. If the record is successfully retrieved, TRUE is returned.

See also at() [p. 117] and isValid() [p. 118].

Examples: sql/overview/basicbrowsing/main.cpp, sql/overview/basicbrowsing2/main.cpp, sql/overview/retrieve1/main.cpp, sql/overview/subclass3/main.cpp, sql/overview/subclass4/main.cpp, sql/overview/subclass5/main.cpp and sql/sqltable/main.cpp.

int QSqlQuery::numRowsAffected () const

Returns the number of rows affected by the result's SQL statement, or -1 if it cannot be determined. Note that for SELECT statements, this value will be the same as size(). If the query is not active (isActive() returns FALSE), -1 is returned.

See also size() [p. 120] and QSqlDriver::hasFeature() [p. 89].

Examples: sql/overview/basicbrowsing2/main.cpp and sql/overview/basicdatamanip/main.cpp.

QSqlQuery & QSqlQuery::operator= (const QSqlQuery & other)

Assigns *other* to the query.

bool QSqlQuery::prev () [virtual]

Retrieves the previous record in the result, if available, and positions the query on the retrieved record. Note that the result must be in an active state and isSelect() must return TRUE before calling this function or it will do nothing and return FALSE.

The following rules apply:

- If the result is currently located before the first record, there is no change and FALSE is returned.
- If the result is currently located after the last record, an attempt is made to retrieve the last record.
- If the result is somewhere in the middle, an attempt is made to retrieve the previous record.

If the record could not be retrieved, the result is positioned before the first record and FALSE is returned. If the record is successfully retrieved, TRUE is returned.

See also at() [p. 117].

const QSqlResult * QSqlQuery::result () const

Returns a pointer to the result associated with the query.

bool QSqlQuery::seek (int i, bool relative = FALSE) [virtual]

Retrieves the record at position (or offset) *i*, if available, and positions the query on the retrieved record. The first record is at position zero. Note that the query must be in an active state and isSelect() must return TRUE before calling this function.

The following rules apply:

If *relative* is FALSE (the default), the following rules apply:

- If *i* is negative, the result is positioned before the first record and FALSE is returned.
- Otherwise, an attempt is made to move to the record at position *i*. If the record at position *i* could not be retrieved, the result is positioned after the last record and FALSE is returned. If the record is successfully retrieved, TRUE is returned.

If *relative* is TRUE, the following rules apply:

- If the result is currently positioned before the first record or on the first record, and *i* is negative, there is no change, and FALSE is returned.
- If the result is currently located after the last record, and *i* is positive, there is no change, and FALSE is returned.
- If the result is currently located somewhere in the middle, and the relative offset *i* moves the result below zero, the result is positioned before the first record and FALSE is returned.
- Otherwise, an attempt is made to move to the record *i* records ahead of the current record (or *i* records behind the current record if *i* is negative). If the record at offset *i* could not be retrieved, the result is positioned after the last record if $i >= 0$, (or before the first record if *i* is negative), and FALSE is returned. If the record is successfully retrieved, TRUE is returned.

Example: sql/overview/navigating/main.cpp.

int QSqlQuery::size () const

Returns the size of the result, (number of rows returned), or -1 if the size cannot be determined or the database does not support reporting information about query sizes. Note that for non-SELECT statements (isSelect() returns FALSE), size() will return -1. If the query is not active (isActive() returns FALSE), -1 is returned.

To determine the number of rows affected by a non-SELECT statement, use numRowsAffected().

See also isActive() [p. 118], numRowsAffected() [p. 119] and QSqlDriver::hasFeature() [p. 89].

Example: sql/overview/navigating/main.cpp.

QVariant QSqlQuery::value (int i) const [virtual]

Returns the value of field *i* (zero based).

The fields are numbered from left to right using the text of the `SELECT` statement, e.g. in "select forename, surname from people;", field 0 is forename and field 1 is surname. Using `SELECT *` is not recommended because the order of the fields in the query is undefined.

An invalid QVariant is returned if field *i* does not exist, if the query is inactive, or if the query is positioned on an invalid record.

See also prev() [p. 119], next() [p. 118], first() [p. 117], last() [p. 118], seek() [p. 120], isActive() [p. 118] and isValid() [p. 118].

Examples: sql/overview/basicbrowsing/main.cpp, sql/overview/basicbrowsing2/main.cpp, sql/overview/retrieve1/main.cpp, sql/overview/subclass3/main.cpp, sql/overview/subclass4/main.cpp, sql/overview/subclass5/main.cpp and sql/sqltable/main.cpp.

QSqlRecord Class Reference

The QSqlRecord class encapsulates a database record, i.e. a set of database fields.

This class is part of the **sql** module.

`#include <qsqlrecord.h>`

Inherited by QSqlCursor [p. 67] and QSqlIndex [p. 109].

Public Members

- **QSqlRecord** ()
- **QSqlRecord** (const QSqlRecord & other)
- QSqlRecord & **operator=** (const QSqlRecord & other)
- virtual **~QSqlRecord** ()
- virtual QVariant **value** (int i) const
- virtual QVariant **value** (const QString & name) const
- virtual void **setValue** (int i, const QVariant & val)
- virtual void **setValue** (const QString & name, const QVariant & val)
- bool **isGenerated** (int i) const
- bool **isGenerated** (const QString & name) const
- virtual void **setGenerated** (const QString & name, bool generated)
- virtual void **setGenerated** (int i, bool generated)
- virtual void **setNull** (int i)
- virtual void **setNull** (const QString & name)
- bool **isNull** (int i)
- bool **isNull** (const QString & name)
- int **position** (const QString & name) const
- QString **fieldName** (int i) const
- QSqlField * **field** (int i)
- QSqlField * **field** (const QString & name)
- const QSqlField * **field** (int i) const
- const QSqlField * **field** (const QString & name) const
- virtual void **append** (const QSqlField & field)
- virtual void **insert** (int pos, const QSqlField & field)
- virtual void **remove** (int pos)
- bool **isEmpty** () const
- bool **contains** (const QString & name) const
- virtual void **clear** ()
- virtual void **clearValues** (bool nullify = FALSE)
- uint **count** () const
- virtual QString **toString** (const QString & prefix = QString::null, const QString & sep = ",") const
- virtual QStringList **toStringList** (const QString & prefix = QString::null) const

Detailed Description

The QSqlRecord class encapsulates a database record, i.e. a set of database fields.

The QSqlRecord class encapsulates the functionality and characteristics of a database record (usually a table or view within the database). QSqlRecords support adding and removing fields as well as setting and retrieving field values.

QSqlRecord is implicitly shared. This means you can make copies of the record in time O(1). If multiple QSqlRecord instances share the same data and one is modifying the record's data then this modifying instance makes a copy and modifies its private copy - thus it does not affect other instances.

See also Database Classes.

Member Function Documentation

QSqlRecord::QSqlRecord ()

Constructs an empty record.

QSqlRecord::QSqlRecord (const QSqlRecord & other)

Constructs a copy of *other*.

QSqlRecord::~QSqlRecord () [virtual]

Destroys the object and frees any allocated resources.

void QSqlRecord::append (const QSqlField & field) [virtual]

Append a copy of field *field* to the end of the record.

Reimplemented in QSqlIndex.

void QSqlRecord::clear () [virtual]

Removes all fields from the record.

See also clearValues() [p. 123].

Reimplemented in QSqlCursor.

void QSqlRecord::clearValues (bool nullify = FALSE) [virtual]

Clears the value of all fields in the record. If *nullify* is TRUE, (it's default is FALSE), each field is set to null.

bool QSqlRecord::contains (const QString & name) const

Returns TRUE if there is a field in the record called *name*, otherwise returns FALSE.

uint QSqlRecord::count () const

Returns the number of fields in the record.

QSqlField * QSqlRecord::field (int i)

Returns a pointer to the field at position *i* within the record, or 0 if it cannot be found.

QSqlField * QSqlRecord::field (const QString & name)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a pointer to the field with name *name* within the record, or 0 if it cannot be found. Field names are not case-sensitive.

const QSqlField * QSqlRecord::field (int i) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

const QSqlField * QSqlRecord::field (const QString & name) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a pointer to the field with name *name* within the record, or 0 if it cannot be found. Field names are not case-sensitive.

QString QSqlRecord::fieldName (int i) const

Returns the name of the field at position *i*. If the field does not exist, QString::null is returned.

void QSqlRecord::insert (int pos, const QSqlField & field) [virtual]

Insert a copy of *field* at position *pos*. If a field already exists at *pos*, it is removed.

bool QSqlRecord::isEmpty () const

Returns TRUE if there are no fields in the record, otherwise returns FALSE.

bool QSqlRecord::isGenerated (const QString & name) const

Returns TRUE if the field *name* is to be generated (the default), otherwise returns FALSE. If the field does not exist, FALSE is returned.

See also setGenerated() [p. 125].

bool QSqlRecord::isGenerated (int i) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the field with the index *i* is to be generated (the default), otherwise returns FALSE. If the field does not exist, FALSE is returned.

See also setGenerated() [p. 125].

bool QSqlRecord::isNull (const QString & name)

Returns TRUE if the field *name* is currently null, otherwise returns FALSE. If the field *name* doesn't exist the return value is TRUE.

See also position() [p. 125].

bool QSqlRecord::isNull (int i)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the field *i* is currently null, otherwise returns FALSE. If the index *i* doesn't exist the return value is TRUE.

See also fieldName() [p. 124].

QSqlRecord & QSqlRecord::operator= (const QSqlRecord & other)

Sets the record equal to *other*.

int QSqlRecord::position (const QString & name) const

Returns the position of the field named *name* within the record, or -1 if it cannot be found. Field names are not case-sensitive. If more than one field matches, the first one is returned.

void QSqlRecord::remove (int pos) [virtual]

Removes the field at *pos*. If *pos* does not exist, nothing happens.

Reimplemented in QSqlCursor.

void QSqlRecord::setGenerated (const QString & name, bool generated) [virtual]

Sets the generated flag for the field *name* to *generated*. If the field does not exist, nothing happens. Only fields that have *generated* set to TRUE are included in the SQL that is generated, e.g. by QSqlCursor.

See also isGenerated() [p. 124].

Reimplemented in QSqlCursor.

void QSqlRecord::setGenerated (int i, bool generated) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the generated flag for the field *i* to *generated*.

See also isGenerated() [p. 124].

Reimplemented in QSqlCursor.

void QSqlRecord::setNull (int i) [virtual]

Sets the value of field *i* to NULL. If the field does not exist, nothing happens.

void QSqlRecord::setNull (const QString & name) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the value of field *name* to NULL. If the field does not exist, nothing happens.

void QSqlRecord::setValue (int i, const QVariant & val) [virtual]

Sets the value of the field at position *i* to *val*. If the field does not exist, nothing happens.

Examples: sql/overview/extract/main.cpp, sql/overview/insert/main.cpp, sql/overview/insert2/main.cpp, sql/overview/order2/main.cpp, sql/overview/subclass5/main.cpp, sql/overview/update/main.cpp and sql/sqltable/main.cpp.

void QSqlRecord::setValue (const QString & name, const QVariant & val) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the value of field *name* to *val*. If the field does not exist, nothing happens.

QString QSqlRecord::toString (const QString & prefix = QString::null, const QString & sep = ",") const [virtual]

Returns a list of all the record's field names as a string separated by *sep*.

Note that fields which are not generated are *not* included (see isGenerated()). The returned string is suitable, for example, for generating SQL SELECT statements. If a *prefix* is specified, e.g. a table name, all fields are prefixed in the form:

"*prefix*. <fieldname>"

QStringList QSqlRecord::toStringList (const QString & prefix = QString::null) const [virtual]

Returns a list of all the record's field names, each having the prefix *prefix*.

Note that fields which have generated set to FALSE are *not* included. (See isGenerated()). If *prefix* is supplied, e.g. a table name, all fields are prefixed in the form:

"*prefix*. <fieldname>"

QVariant QSqlRecord::value (int i) const [virtual]

Returns the value of the field located at position *i* in the record. If field *i* does not exist the resultant behaviour is undefined.

This function should be used with QSqlQuerys. When working with a QSqlCursor the value(const QString&) overload which uses field names is more appropriate.

Example: sql/overview/update/main.cpp.

QVariant QSqlRecord::value (const QString & name) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the value of the field named *name* in the record. If field *name* does not exist the resultant behaviour is undefined.

QSqlRecordInfo Class Reference

The API for this class is under development and is subject to change.
We do not recommend the use of this class for production work at this time.

The QSqlRecordInfo class encapsulates a set of database field meta data.

This class is part of the **sql** module.

```
#include <qsqlrecord.h>
```

Public Members

- **QSqlRecordInfo** ()
- **QSqlRecordInfo** (const QSqlFieldInfoList & other)
- **QSqlRecordInfo** (const QSqlRecord & other)
- size_type **contains** (const QString & fieldName) const
- QSqlFieldInfo **find** (const QString & fieldName) const
- QSqlRecord **toRecord** () const

Detailed Description

The QSqlRecordInfo class encapsulates a set of database field meta data.

This class is a QValueList that holds a set of database field meta data. Use contains() to see if a given field name exists in the record, and use find() to get a QSqlFieldInfo record for a named field.

See also QValueList [Datastructures and String Handling with Qt], QSqlFieldInfo [p. 101] and Database Classes.

Member Function Documentation

QSqlRecordInfo::QSqlRecordInfo ()

Constructs an empty recordinfo object

QSqlRecordInfo::QSqlRecordInfo (const QSqlFieldInfoList & other)

Constructs a copy of *other*.

QSqlRecordInfo::QSqlRecordInfo (const QSqlRecord & other)

Constructs a QSqlRecordInfo object based on the fields in the QSqlRecord *other*.

size_type QSqlRecordInfo::contains (const QString & fieldName) const

Returns the number of times a field named *fieldName* occurs in the record. Returns 0 if no field by that name could be found.

QSqlFieldInfo QSqlRecordInfo::find (const QString & fieldName) const

Returns a QSqlFieldInfo object for the first field in the record which has the field name *fieldName*. If no matching field is found then an empty QSqlFieldInfo object is returned.

QSqlRecord QSqlRecordInfo::toRecord () const

Returns an empty QSqlRecord based on the field information in this QSqlRecordInfo.

QSqlResult Class Reference

The QSqlResult class provides an abstract interface for accessing data from SQL databases.

This class is part of the **sql** module.

`#include <qsqlresult.h>`

Public Members

- virtual **~QSqlResult** ()

Protected Members

- **QSqlResult** (const QSqlDriver * db)
- int **at** () const
- QString **lastQuery** () const
- QSqlError **lastError** () const
- bool **isValid** () const
- bool **isActive** () const
- bool **isSelect** () const
- bool **isForwardOnly** () const
- const QSqlDriver * **driver** () const
- virtual void **setAt** (int at)
- virtual void **setActive** (bool a)
- virtual void **setLastError** (const QSqlError & e)
- virtual void **setQuery** (const QString & query)
- virtual void **setSelect** (bool s)
- virtual void **setForwardOnly** (bool forward)
- virtual QVariant **data** (int i)
- virtual bool **isNull** (int i)
- virtual bool **reset** (const QString & query)
- virtual bool **fetch** (int i)
- virtual bool **fetchNext** ()
- virtual bool **fetchPrev** ()
- virtual bool **fetchFirst** ()
- virtual bool **fetchLast** ()
- virtual int **size** ()
- virtual int **numRowsAffected** ()

Detailed Description

The QSqlResult class provides an abstract interface for accessing data from SQL databases.

Normally you would use QSqlQuery instead of QSqlResult since QSqlQuery provides a generic wrapper for database-specific implementations of QSqlResult.

See also QSql [p. 65] and Database Classes.

Member Function Documentation

QSqlResult::QSqlResult (const QSqlDriver * db) [protected]

Protected constructor which creates a QSqlResult using database *db*. The object is initialized to an inactive state.

QSqlResult::~QSqlResult () [virtual]

Destroys the object and frees any allocated resources.

int QSqlResult::at () const [protected]

Returns the current (zero-based) position of the result.

QVariant QSqlResult::data (int i) [virtual protected]

Returns the data for field *i* (zero-based) as a QVariant. This function is only called if the result is in an active state and is positioned on a valid record and *i* is non-negative. Derived classes must reimplement this function and return the value of field *i*, or QVariant() if it cannot be determined.

const QSqlDriver * QSqlResult::driver () const [protected]

Returns the driver associated with the result.

bool QSqlResult::fetch (int i) [virtual protected]

Positions the result to an arbitrary (zero-based) index *i*. This function is only called if the result is in an active state. Derived classes must reimplement this function and position the result to the index *i*, and call setAt() with an appropriate value. Return TRUE to indicate success, FALSE for failure.

bool QSqlResult::fetchFirst () [virtual protected]

Positions the result to the first record in the result. This function is only called if the result is in an active state. Derived classes must reimplement this function and position the result to the first record, and call setAt() with an appropriate value. Return TRUE to indicate success, FALSE for failure.

bool QSqlResult::fetchLast () [virtual protected]

Positions the result to the last record in the result. This function is only called if the result is in an active state. Derived classes must reimplement this function and position the result to the last record, and call setAt() with an appropriate value. Return TRUE to indicate success, FALSE for failure.

bool QSqlResult::fetchNext () [virtual protected]

Positions the result to the next available record in the result. This function is only called if the result is in an active state. The default implementation calls fetch() with the next index. Derived classes can reimplement this function and position the result to the next record in some other way, and call setAt() with an appropriate value. Return TRUE to indicate success, FALSE for failure.

bool QSqlResult::fetchPrev () [virtual protected]

Positions the result to the previous available record in the result. This function is only called if the result is in an active state. The default implementation calls fetch() with the previous index. Derived classes can reimplement this function and position the result to the next record in some other way, and call setAt() with an appropriate value. Return TRUE to indicate success, FALSE for failure.

bool QSqlResult::isActive () const [protected]

Returns TRUE if the result has records to be retrieved, otherwise returns FALSE.

bool QSqlResult::isForwardOnly () const [protected]

Returns TRUE when you can only scroll forward through a result set otherwise FALSE

bool QSqlResult::isNull (int i) [virtual protected]

Returns TRUE if the field at position *i* is NULL, otherwise returns FALSE.

bool QSqlResult::isSelect () const [protected]

Returns TRUE if the current result is from a SELECT statement, otherwise returns FALSE.

bool QSqlResult::isValid () const [protected]

Returns TRUE if the result is positioned on a valid record (that is, the result is not positioned before the first or after the last record); otherwise returns FALSE.

QSqlError QSqlResult::lastError () const [protected]

Returns the last error associated with the result.

QString QSqlResult::lastQuery () const [protected]

Returns the current SQL query text, or QString::null if there is none.

int QSqlResult::numRowsAffected () [virtual protected]

Returns the number of rows affected by the last query executed.

bool QSqlResult::reset (const QString & query) [virtual protected]

Sets the result to use the SQL statement *query* for subsequent data retrieval. Derived classes must reimplement this function and apply the *query* to the database. This function is called only after the result is set to an inactive state and is positioned before the first record of the new result. Derived classes should return TRUE if the query was successful and ready to be used, FALSE otherwise.

void QSqlResult::setActive (bool a) [virtual protected]

Protected function provided for derived classes to set the internal active state to the value of *a*.

See also isActive() [p. 132].

void QSqlResult::setAt (int at) [virtual protected]

Protected function provided for derived classes to set the internal (zero-based) result index to *at*.

See also at() [p. 131].

void QSqlResult::setForwardOnly (bool forward) [virtual protected]

Sets forward only mode to *forward*. If forward is TRUE only fetchNext() is allowed for navigating the results. Forward only mode needs far less memory since results do not have to be cached. forward only mode is off by default.

See also fetchNext() [p. 132].

void QSqlResult::setLastError (const QSqlError & e) [virtual protected]

Protected function provided for derived classes to set the last error to the value of *e*.

See also lastError() [p. 132].

void QSqlResult::setQuery (const QString & query) [virtual protected]

Sets the current query for the result to *query*. The result must be reset() in order to execute the query on the database.

void QSqlResult::setSelect (bool s) [virtual protected]

Protected function provided for derived classes to indicate whether or not the current statement is an SQL SELECT statement. The *s* parameter should indicate TRUE if the statement is a SELECT statement, otherwise FALSE.

int QSqlResult::size () [virtual protected]

Returns the size of the result or -1 if it cannot be determined.

Index