

Lecture SQL06

Qt and SQL – Part I

Outline

- Accessing SQL from Qt
- Connecting to MySQL
- Connecting to SQLite
- Connection Errors
- Querying the Database from Qt
- Interacting with the Database via GUI

Accessing SQL from Qt

- Modify the **.pro** file
 - As with the networks module, you must modify the **.pro** file for your Qt program to access the Qt SQL module
 - Add the following statement

QT += sql

- Include the header file in your source file(s)

#include <QtSql>

Connecting to MySQL

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

db.setHostName( "localhost" );    // db on local machine
db.setPort(3300);                 // port number
db.setDatabaseName( "somefilenamehere" );
db.setUserName( "myusername" );
db.setPassword( "mypassword" );

if ( !db.open() )
{
    qDebug() << db.lastError();
    qDebug() << "Error:  Unable to connect due to above error";
}
```

Connecting to SQLite

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QSQLITE" );  
  
db.setDatabaseName( "vetclinic.db" );  
  
if ( !db.open() )  
{  
    qDebug() << db.lastError();  
    qDebug() << "Error:  Unable to connect due to above error";  
}
```

Can also specify

```
db.setDatabaseName( ":memory:" );
```

to have database created in memory

Connection Errors - 1

- In general your program may be interacting with a remote database
- As such, a query may not complete successfully for a variety of reasons
 - Connection failure (ex. file not found)
 - Authentication problem
 - Network issue
 - Poorly formed query
 - Etc.

Connection Errors - 2

- You should add error handling code
 - To detect problems related to establishing an initial connection
 - To address the possibility that any subsequent database access may also fail

Connecting to SQLite

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QSQLITE" );  
  
db.setDatabaseName( "vetclinic.db" );  
  
if ( !db.open() )  
{  
    qDebug() << db.lastError();  
    qDebug() << "Error:  Unable to connect due to above error";  
}
```

Can also specify

```
db.setDatabaseName( ":memory:" );
```

to have database created in memory

Querying the Database from Qt Without a GUI

customers

UID	Last Name	First Name
128	Smith	John
324	Doe	John
245	Jones	Mark
756	Smith	Jane
459	Moore	Sara
721	Parks	Ralph

vets

UID
324
245

accounts

UID	Balance
128	0
756	45
459	0
721	10

Relations

pets

UID	Pet Name	Type
128	Spot	Dog
324	Rex	Dog
756	Tiger	Cat
756	Fluffy	Cat
459	Tweety	Bird
721	Yippy	Dog
128	Rover	Dog
245	Stripes	Cat
324	Cupcake	Dog
459	Chewy	Dog

VetClinic Qt/SQL Example - 1

```
//  
// Direct Interaction with an SQLite Database from Qt4  
//  
#include <QApplication>  
#include <QtSql>  
#include <QtDebug>  
  
int main(int argc, char* argv[])  
{  
    QApplication myApp(argc, argv);  
  
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");  
    db.setDatabaseName("vetclinic.db");  
  
    if ( !db.open() )  
    {  
        qDebug() << db.lastError();  
        qDebug() << "Error: Unable to connect";  
        return 1;  
    }  
}
```

VetClinic Qt/SQL Example - 2

```
// Direct Interaction with an SQLite Database from Qt4
```

```
// Create query and use exec call to hand it an argument and execute it  
QSqlQuery q;
```

```
q.exec("SELECT * FROM pets WHERE type='Dog'"); <== No error handling here
```

```
while ( q.next() )  
{  
    qDebug() << "UID=" << q.value(0).toInt()  
        << "    PetName=" << q.value(1).toString()  
        << "    Type=" << q.value(2).toString();  
}
```

exec() returns true if query executes successfully; false otherwise

next() / previous() retrieves next/previous record in the result

Output

UID= 128	PetName= "Spot"	Type= "Dog"
UID= 324	PetName= "Rex"	Type= "Dog"
UID= 721	PetName= "Yippy"	Type= "Dog"
UID= 128	PetName= "Rover"	Type= "Dog"
UID= 324	PetName= "Cupcake"	Type= "Dog"
UID= 459	PetName= "Chewy"	Type= "Dog"

UAH
CPE 353

VetClinic Qt/SQL Example - 3

```
// Direct Interaction with an SQLite Database from Qt4
```

```
// Create query object and execute it immediately
```

```
QSqlQuery qq("SELECT * FROM customers WHERE lastname='Smith'");
```

```
if ( !qq.isActive() )      <== Error handling here
```

```
{  
    qDebug() << qq.lastError();  
    qDebug() << "Error: unable to complete query";  
    return 1;  
}
```

isActive() returns true if query has executed successfully
but is not yet finished

```
while ( qq.next() )
```

```
{  
    qDebug() << "UID=" << qq.value(0).toInt()  
        << "    LastName=" << qq.value(1).toString()  
        << "    FirstName=" << qq.value(2).toString();  
}
```

Output

UID= 128	LastName= "Smith"	FirstName= "John"
UID= 756	LastName= "Smith"	FirstName= "Jane"

VetClinic Qt/SQL Example - 4

```
// Direct Interaction with an SQLite Database from Qt4

// Create poorly formed SQL query and attempt to execute it
QSqlQuery qqq("SELECT * FROM goats");

if ( !qqq.isActive() )    <== Error handling here
{
    qDebug() << qqq.lastError();
    qDebug() << "Deliberate failed SQL query";
}
```

Output

```
QSqlError(1, "Unable to execute statement", "no such table: goats")
Deliberate failed SQL query
```

VetClinic Qt/SQL Example - 5

// Direct Interaction with an SQLite Database from Qt4

```
// Create query using placeholders for actual values
// Bind values to the placeholders at runtime
// Execute query
```

```
QSqlQuery qqqq;
```

```
qqqq.prepare("INSERT INTO pets (uid, petname, type) "
             "VALUES (:uid, :petname, :type)");
```

<== Method #1

```
qqqq.bindValue(":uid", 999);
```

```
qqqq.bindValue(":petname", "Jaws");
```

```
qqqq.bindValue(":type", "Shark");
```

```
if ( !qqqq.exec() )
```

```
{
```

```
    qDebug() << qqqq.lastError();
```

```
    qDebug() << "Error on INSERT";
```

```
    return 1;
```

```
}
```

Use of **bindValue** with unvalidated user inputs introduces a risk of SQL injection attack.

VetClinic Qt/SQL Example - 6

// Direct Interaction with an SQLite Database from Qt4

```
// Create query using placeholders for actual values
// Bind values to the placeholders at runtime
// Execute query
```

Method #2

```
QSqlQuery qqqq;
qqqq.prepare("INSERT INTO pets (uid, petname, type) VALUES(?, ?, ?)");
qqqq.addBindValue(999);
qqqq.addBindValue("Jaws");
qqqq.addBindValue("Shark");
if ( !qqqq.exec() )
{
    qDebug() << qqqq.lastError();
    qDebug() << "Error on INSERT";
    return 1;
}
```


VetClinic Qt/SQL Example - 7

// Direct Interaction with an SQLite Database from Qt4

```
QSqlQuery qqqqq("SELECT * FROM pets;");
while ( qqqqq.next() )
{
    qDebug() << "UID=" << qqqqq.value(0).toInt()
               << "    PetName=" << qqqqq.value(1).toString()
               << "    Type=" << qqqqq.value(2).toString();
}
qDebug() << endl;
```

Output

UID= 128	PetName= "Spot"	Type= "Dog"
UID= 324	PetName= "Rex"	Type= "Dog"
UID= 756	PetName= "Tiger"	Type= "Cat"
UID= 756	PetName= "Fluffy"	Type= "Cat"
UID= 459	PetName= "Tweety"	Type= "Bird"
UID= 721	PetName= "Yippy"	Type= "Dog"
UID= 128	PetName= "Rover"	Type= "Dog"
UID= 245	PetName= "Stripes"	Type= "Cat"
UID= 324	PetName= "Cupcake"	Type= "Dog"
UID= 459	PetName= "Chewy"	Type= "Dog"
UID= 999	PetName= "Jaws"	Type= "Shark"

VetClinic SQL Example - 8

```
qqqqq.exec("UPDATE pets SET type='Goldfish' WHERE petname='Jaws'");
qqqqq.exec("SELECT * FROM pets;");
while ( qqqqq.next() )
{
    qDebug() << "UID=" << qqqqq.value(0).toInt()
               << "    PetName=" << qqqqq.value(1).toString()
               << "    Type=" << qqqqq.value(2).toString();
}
qDebug() << endl;
```

Updating a Row

Output:

UID= 128	PetName= "Spot"	Type= "Dog"
UID= 324	PetName= "Rex"	Type= "Dog"
UID= 756	PetName= "Tiger"	Type= "Cat"
UID= 756	PetName= "Fluffy"	Type= "Cat"
UID= 459	PetName= "Tweety"	Type= "Bird"
UID= 721	PetName= "Yippy"	Type= "Dog"
UID= 128	PetName= "Rover"	Type= "Dog"
UID= 245	PetName= "Stripes"	Type= "Cat"
UID= 324	PetName= "Cupcake"	Type= "Dog"
UID= 459	PetName= "Chewy"	Type= "Dog"
UID= 999	PetName= "Jaws"	Type= "Goldfish"

UAH
CPE 353

VetClinic SQL Example - 9

```
qqqqq.exec("DELETE FROM pets WHERE petname='Jaws'");
```

Deleting a Row

```
qqqqq.exec("SELECT * FROM pets;");  
while ( qqqqq.next() )  
{  
    qDebug() << "UID=" << qqqqq.value(0).toInt()  
        << "    PetName=" << qqqqq.value(1).toString()  
        << "    Type=" << qqqqq.value(2).toString();  
}  
qDebug() << endl;
```

Output:

UID= 128	PetName= "Spot"	Type= "Dog"
UID= 324	PetName= "Rex"	Type= "Dog"
UID= 756	PetName= "Tiger"	Type= "Cat"
UID= 756	PetName= "Fluffy"	Type= "Cat"
UID= 459	PetName= "Tweety"	Type= "Bird"
UID= 721	PetName= "Yippy"	Type= "Dog"
UID= 128	PetName= "Rover"	Type= "Dog"
UID= 245	PetName= "Stripes"	Type= "Cat"
UID= 324	PetName= "Cupcake"	Type= "Dog"
UID= 459	PetName= "Chewy"	Type= "Dog"

UAH
CPE 353

Querying the Database from Qt With a GUI

Models

- **QSqlQueryModel**
 - Read-only model for displaying SELECT output
- **QSqlTableModel**
 - Editable model for single table
- **QSqlRelationalModel**
 - Editable model for single table which refers to other tables

VetClinic SQL Example - 11

```
// QSqlQueryModel
```

```
#include <QApplication>
#include <QtSql>
#include <QTableView>
#include <QtDebug>
using namespace std;
```

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

```
    QApplication myApp(argc, argv);
```

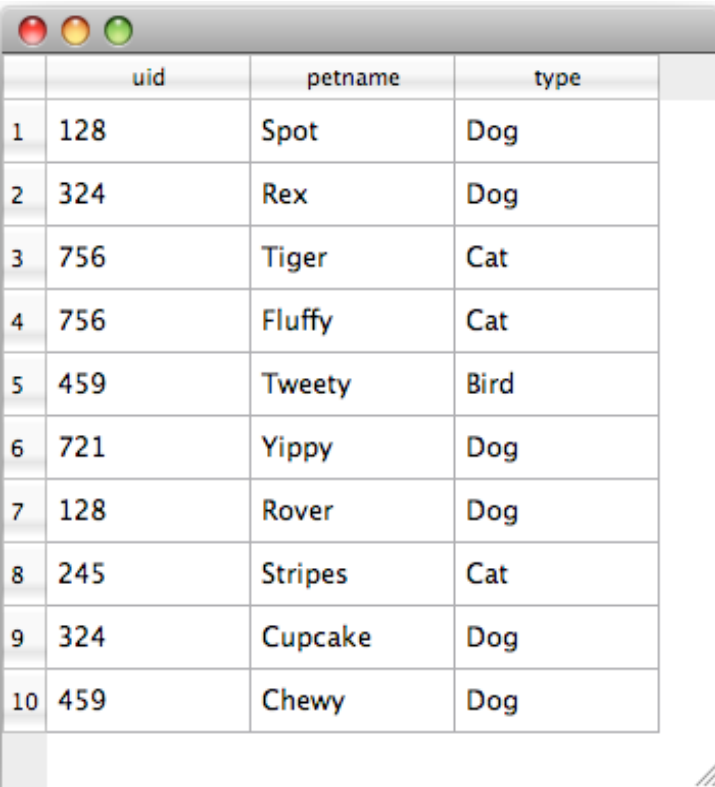
```
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("vetclinic.db");
```

```
    if ( !db.open() )
    {
        qDebug() << db.lastError();
        qDebug() << "Error: Unable to connect";
        return 1;
    }
```

```
    QSqlQueryModel model;
    model.setQuery("SELECT * FROM pets");
```

```
    QTableView view;
    view.setModel(&model);
    view.show();
```

```
    return myApp.exec();
} // End main()
```



A screenshot of a Qt application window displaying a QTableView. The window has a standard macOS-style title bar with red, yellow, and green buttons. The table contains 10 rows of pet data, with columns for an index, uid, petname, and type. The data is as follows:

	uid	petname	type
1	128	Spot	Dog
2	324	Rex	Dog
3	756	Tiger	Cat
4	756	Fluffy	Cat
5	459	Tweety	Bird
6	721	Yippy	Dog
7	128	Rover	Dog
8	245	Stripes	Cat
9	324	Cupcake	Dog
10	459	Chewy	Dog

VetClinic SQL Example - 12

```
// QSqlQueryModel
#include <QApplication>
#include <QtSql>
#include <QTableView>
#include <QtDebug>
using namespace std;

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

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

    if ( !db.open() )
    {
        qDebug() << db.lastError();
        qDebug() << "Error: Unable to connect";
        return 1;
    }

    QSqlQuery q("INSERT INTO pets (uid, petname, type) VALUES(999, 'Jaws', 'Shark')");
    QSqlTableModel* model = new QSqlTableModel;
    model->setTable("pets");
    model->select();
    model->setEditStrategy(QSqlTableModel::OnRowChange);

    QTableView* view = new QTableView;
    view->setModel(model);
    view->show();

    return myApp.exec();
} // End main()
```

VetClinic SQL Example - 13



	uid	petname	type
1	128	Spot	Dog
2	324	Rex	Dog
3	756	Tiger	Cat
4	756	Fluffy	Cat
5	459	Tweety	Bird
6	721	Yippy	Dog
7	128	Rover	Dog
8	245	Stripes	Cat
9	324	Cupcake	Dog
10	459	Chewy	Dog
11	999	Jaws	Shark



	uid	petname	type
1	128	Spot	Dog
2	324	Rex	Dog
3	756	Tiger	Cat
4	756	Fluffy	Cat
5	459	Tweety	Bird
6	721	Yippy	Dog
7	128	Rover	Dog
8	245	Stripes	Cat
9	324	Cupcake	Dog
10	459	Chewy	Dog
11	999	Orca	Shark

VetClinic SQL Example - 14

```
// QSqlQueryModel
#include <QApplication>
#include <QtSql>
#include <QTableView>
#include <QtDebug>
using namespace std;

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

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

    if ( !db.open() )
    {
        qDebug() << db.lastError();
        qDebug() << "Error: Unable to connect";
        return 1;
    }

    QSqlQuery q("INSERT INTO pets (uid, petname, type) VALUES(999, 'Jaws', 'Shark')");
    QSqlTableModel* model = new QSqlTableModel;
    model->setTable("pets");
    model->select();
    model->setEditStrategy(QSqlTableModel::OnFieldChange);

    QTableView* view = new QTableView;
    view->setModel(model);
    view->show();

    return myApp.exec();
} // End main()
```