**PROGRAMACIÓN APLICADA A LA AUTOMATIZACIÓN**

**PREPRÁCTICA # 12 Manejo de Bases de Datos.**

**CAPÍTULO DEL CURSO:** Comunicación y manejo de datos en automatización.

**TEMA DE LA ACTIVIDAD:** Uso de Qt Creator para desarrollar rápidamente aplicaciones que manejen bases de datos.

**NOTA**

**OBJETIVOS DE APRENDIZAJE:**

* Elaborar aplicaciones que permitan manejar Base de Datos.
* Usar el ambiente de desarrollo integrado de Qt Creator para la edición, compilación, ejecución y depuración de aplicaciones visuales sobre Windows.

**DURACIÓN:** 120 minutos

**MATERIALES Y HERRAMIENTAS:**

* Computadora con Qt Creator.

**INTRODUCCIÓN**

El almacenaje de información de forma que sea más fácil encontrar lo que se quiere de ella es uno de los grandes problemas actuales, principalmente en estos tiempos que se avecinan de internet de las cosas, computación en la nube y donde cada vez es mayor la cantidad de datos recolectados. Una de las principales herramientas para ayudar a solucionar estos y muchos otros problemas son las bases de datos. En el caso de las bases de datos relacionales son ampliamente utilizadas para la recolección de información en los elementos de la informática industrial como pueden ser los SCADA, los sistemas MES y los sistemas ERP. También son utilizadas por muchos otros sistemas informáticos contemporáneos. El DBMS más utilizado en este modelo de BD es el SQLite por su poco consumo de espacio y su facilidad para compartir el mismo espacio con cualquier aplicación Web o de escritorio. Qt permite manejar con facilidad las BD conteniendo diversos drivers para la variedad de tipos y diversas clases para su manejo.

En esta parte, el estudiante debe crear una aplicación que permita la lectura y actualización de una base de datos relacional de SQLite. Se debe emplear como bibliografía la conferencia XIII o sesión de clase 13 y la propia bibliografía allí propuesta.

**DESCRIPCIÓN DE LA PRÁCTICA:** El estudiante debe realizar la práctica siguiendo las instrucciones o guías planteadas a continuación, realizando cada ejercicio (creación de la BD y el manejo de la BD ya creada para las funcionalidades de recuperación y actualización) de forma independiente. Al finalizar debe entregar los archivos proyecto realizados compactados para verificar su cumplimiento.

**PROCEDIMIENTO:**

**Guía de la práctica:**

En la sesión de clase teórica se revisaron las definiciones relacionadas a BD (mayormente la relacional. También se revisó el lenguaje SQL, y las sentencias principales que nos pueden servir para trabajar con una base de datos. Quedó claro que SQL es un lenguaje universal para realizar operaciones sobre una BD, independientemente del formato o sistema de gestión de BD (SGBD o DBMS) utilizado para crearla o mantenerla operativa.

Cuando no se utiliza un SGBD para realizar las operaciones y consultas sobre una BD y visualizar los resultados, se necesita un mecanismo que nos “conecte” con esa base de datos. Hay varias formas de gestionar esa comunicación. En el caso de Qt, existe un manipulador (driver) que nos “conecta” con esa base de datos y se encarga de la “comunicación” con la misma. Por último, estudiamos que Qt tiene un grupo de clases para trabajar con BD relacionales, y se mostró un ejemplo de una clase de “bajo nivel” (no vincula la BD con ninguna vista o interface en particular), que realiza consultas o ejecuta sentencias SQL sobre una BD (SqlQuery).

Un resumen de las clases que dispone Qt para el trabajo con BD es el siguiente:



Qt tiene incorporados varios Drivers para poder manipular diferentes bases de datos. La siguiente figura refleja los drivers de Qt así como el string que los identifica:

****

También se recuerda que cuando abordamos en clases el lenguaje SQL se mencionó que en general hay sentencias SQL orientadas a crear y/o modificar la estructura de la BD (en particular revisamos el CREATE TABLE) y sentencias que manipulan los registros de una o varias tablas (vimos en clase el INSERT, UPDATE, DELETE y el SELECT).

Es importante tener presente esta diferenciación pues los scripts (o ficheros de consulta) deben elaborarse teniendo en cuenta las operaciones iniciales (casi todas de creación de estructura de la BD + inserción de datos iniciales en la BD) en un script independiente y el resto de las operaciones en otro script o en la medida en que se necesite.

Como una segunda variante se muestra cómo realizar estas mismas operaciones iniciales desde un proyecto de Qt para consola, muy parecido al resuelto en la sesión teórica. No tiene que realizar ambas variantes, basta con una de ellas para la creación e inicialización de la base de datos.

En el ejemplo que se desarrollará en esta actividad se utilizará una base de datos que almacena nombre de usuarios y su id en una tabla y los teléfonos de cada usuario en otra tabla:



**Desarrollo.**

**Ejercicio 1: Creación de la BD.**

**Variante 1. Trabajo con el Shell de SQLite (sqlite3.exe).**

En clase ya se mencionó que trabajaríamos con SQLite. Aquí seguiremos la convención comentada ya en la introducción, o sea, realizaremos un primer script para crear la estructura de la BD y para insertar algunos datos, lo que ilustrará además el trabajo con una herramienta bastante simple (y primitiva) pero efectiva cuando se quiere crear la BD y definir su estructura a través de un Script.

Las tablas del ejemplo se definen mediante las siguientes sentencias SQL (**copie las sentencias y almacénelas en el fichero create\_struct.sql en su directorio de trabajo**):

|  |
| --- |
| ATTACH DATABASE "bd\_ej.db" AS bdQt;CREATE TABLE IF NOT EXISTS bdQt.usr (id\_usr INTEGER PRIMARY KEY ASC,n\_usr TEXT);CREATE TABLE IF NOT EXISTS bdQt.phone (id\_phone INTEGER PRIMARY KEY ASC,id\_usr INTEGER, phone INTEGER, FOREIGN KEY(id\_usr) REFERENCES usr(id\_usr));INSERT INTO usr(id\_usr,n\_usr) VALUES (1,"A");INSERT INTO usr(id\_usr,n\_usr) VALUES (2,"B");INSERT INTO usr(id\_usr,n\_usr) VALUES (3,"C");INSERT INTO usr(id\_usr,n\_usr) VALUES (4,"D");INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (1,1,1234567);INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (2,1,1134567);INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (3,2,2234567);INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (4,3,3234567);INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (5,4,4234567);INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (6,3,3334567); |

La primera línea del script anterior asume que ya existe una BD llamada bd\_ej.db que se tratará con el alias bdQt. En breve se verá cómo crear la BD desde el Shell, cuestión que por ser fácil no se incluye en el script. Las líneas 2 y 3 crean las dos tablas que ya con anterioridad se mostraron, y las líneas de la 4 a la 13 insertan elementos en cada una de esas tablas.

**¿Cómo ejecutar el script?** Para esto se hará uso del Shell de sqlite. Este Shell se puede descargar de internet (se encuentra adjunto en para que no tenga que buscarlo), y es básicamente un programa en modo consola que acepta comandos de diverso tipo, entre los cuales está la ejecución de scripts contenidos en ficheros. Veamos su utilización:

• Se abre una sesión en el Shell (consola) de Windows (abra el Shell ejecutando **cmd** en Windows) y nos posicionamos en el directorio de nuestro interés (copie antes la dirección o camino de su carpeta del proyecto y en el Shell cambie de directorio con: cd [dirección], ejemplo: cd C:\Users\alexa\Documents\BD\_Visual).



En dicho directorio debe de haber copiado previamente el Shell de sqlite, que se llama **sqlite3.exe** y el fichero antes creado **create\_struct.sql**.

El Shell de sqlite se puede llamar pasándole como parámetro el nombre de la base de datos que debe abrir cuando se cargue. Si dicha base de datos no existe o no la encuentra en el subdirectorio desde el cual se ejecuta, creará una nueva BD. En nuestro caso debemos poner **sqlite3 bd\_ej.db;** obteniéndose:



Para ver una pequeña ayuda del Shell de sqlite se puede introducir el comando **.help (compruébelo)**. Este Shell ya cargado puede servir para ejecutar directamente sentencias SQL, o para ejecutar las sentencias SQL contenidas en un fichero. En nuestro caso debemos ejecutar las sentencias contenidas en el fichero que se salvó previamente (create\_struct.sql). Para ello usaremos el comando **.read** que necesita como argumento el nombre del fichero que contiene las sentencias sql, por lo que el comando quedaría: **.read create\_struct.sql**



Si se quiere comprobar el resultado de la ejecución del script anterior puede poner una sentencia sql directamente en la línea de comandos que muestre el contenido de la tabla usr y de la tabla phone (**select \* from usr;**) y (**select \* from phone;**). A continuación, se muestra el resultado:



Para salir del sqlite3 se introduce el comando **.quit** con lo que se concluye el primer ejercicio, la creación de la BD y sus tablas.

**Ejercicio 1. 2da Variante. Crear la tabla e inicializarla desde una aplicación de Qt.**

En lugar de haber realizado la primera variante puede crear su tabla y llenarla desde una aplicación de Qt. Para ello crearemos un proyecto Qt en modo consola, le pondremos un nombre acorde y finalizaremos el asistente.



En el fichero .pro de todo proyecto de Qt que maneje bases de datos se debe agregar la línea: **QT += sql**



A continuación, se muestra el código de la aplicación que es muy parecido al explicado en la sesión teórica:

#include <QCoreApplication>

#include <QtSql/QSqlDatabase>

#include <QtSql/QSqlQuery>

#include <QtSql/QSqlError>

#include <QDebug>

#include <iostream>

#include <QString>

using namespace std;

int **main**(int argc, char \*argv[])

{

 QCoreApplication a(*argc*, argv);

 if (!QSqlDatabase::isDriverAvailable("QSQLITE"))

 {

 cout<<"No hay driver QSQLITE!";

 return 0;

 }

 else cout<<"Esta el driver disponible.";

 QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");

 db.setDatabaseName("C:/Users/alexa/Documents/practica14ej1/bd\_ej.db");

 if (!db.open()) {

 qDebug()<<db.lastError().text();

 return 0;

 }

 cout<<"Conectado...\n";

 QSqlQuery query;

 if (!query.exec("CREATE TABLE IF NOT EXISTS usr "

 "(id\_usr INTEGER NOT NULL PRIMARY KEY ASC, "

 "n\_usr VARCHAR(255) NOT NULL)"))

 qDebug()<<query.lastError().text();

 if (!query.exec("CREATE TABLE IF NOT EXISTS phone "

 "(id\_phone INTEGER PRIMARY KEY ASC,id\_usr INTEGER, "

 "phone INTEGER, FOREIGN KEY(id\_usr) REFERENCES usr(id\_usr))"))

 qDebug()<<query.lastError().text();

 query.exec("INSERT INTO usr(id\_usr,n\_usr) VALUES (1,'A')");

 query.exec("INSERT INTO usr(id\_usr,n\_usr) VALUES (2,'B')");

 query.exec("INSERT INTO usr(id\_usr,n\_usr) VALUES (3,'C')");

 query.exec("INSERT INTO usr(id\_usr,n\_usr) VALUES (4,'D')");

 query.exec("INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (1,1,1234567)");

 query.exec("INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (2,1,1134567)");

 query.exec("INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (3,2,2234567)");

 query.exec("INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (4,3,3234567)");

 query.exec("INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (5,4,4234567)");

 query.exec("INSERT INTO phone(id\_phone,id\_usr,phone) VALUES (6,3,3334567)");

 // ejecutando la consulta de usr

 cout<<"Tabla usr:\n";

 QString SqlStmt = "SELECT \* FROM usr;";

 query.exec(SqlStmt);

 // sacando los valores obtenidos de la consulta

 while (query.next()){

 int id=query.value("id\_usr").toInt();

 QString nombre = query.value("n\_usr").toString();

 cout <<"ID:"<<id

 << " Nombre: "<<nombre.toStdString()<<endl;

 }

 // ejecutando la consulta de phone

 SqlStmt = "SELECT \* FROM phone;";

 query.exec(SqlStmt);

 // sacando los valores obtenidos de la consulta

 cout<<"Tabla phone:\n";

 while (query.next()){

 int idp=query.value("id\_phone").toInt();

 int id=query.value("id\_usr").toInt();

 int phone=query.value("phone").toInt();

 cout <<"ID phone:"<<idp

 <<" ID user:"<<id

 << " Phone: "<<phone<<endl;

 }

 db.close();

 return a.exec();

}

Recuerde que debe modificar:

db.setDatabaseName("C:/Users/alexa/Documents/practica14ej1/bd\_ej.db");

para que se adapte al camino o dirección de su directorio de trabajo.

Fíjese que se hace lo mismo que en la variante 1 sólo que ahora se utiliza como DBMS el driver contenido en Qt: **QSQLITE.** Ejecute la aplicación 1 vez para generar el archivo de la base de datos.



**Ejercicio 2. Creación de una aplicación visual para trabajo con la BD.**

Se trabajará sobre un ejemplo que ilustre algunas clases de Qt que permiten realizar las consultas y visualizarlas. El programa contará con una interfaz de alto nivel para mostrar el contenido de las tablas y los resultados de las consultas, así como permitirá realizar consultas sobre la BD y mostrar los resultados.

Cuando se trabaja con BD es muy frecuente presentar la información asociada con la misma en tablas o grids donde las columnas constituyen los atributos que se quieren mostrar de una determinada tabla de la BD y las filas constituyen los registros que cumplen con las condiciones de esa consulta.

Qt presenta varias clases que pueden representar esas tablas, siendo la clase **QTableView** una de las clases representativas. La información que visualiza esa tabla debe provenir de una fuente de datos (datasource), para lo que Qt también ofrece algunas clases (**QSqlQueryModel**, **QSqlTableModel**, etc.) siendo las clases **QSqlTableModel** (incluye la capacidad de edición (update) de los registros) y **QSqlQueryModel** (también se puede hacer update aquí como consulta) dos de las más representativas. La clase QSqlQueryModel es un modelo de solo lectura, y contiene los registros que resultan de una consulta SQL determinada sobre alguna/algunas tablas que componen la BD con la que se está trabajando. La clase **QSqlTableModel** proporciona un modelo de datos de lectura/escritura (permite modificación) asociado a una tabla de la base de datos, por lo que un cambio en dicha tabla se puede reflejar de manera directa en la tabla. Este último se debe utilizar preferiblemente si se quiere **dar acceso al usuario a modificar los registros de la BD**, mientras que el primero garantiza acceso de solo lectura y la modificación debe hacerse a través de una consulta SQL explícita. Este último caso será el que veremos en este ejercicio.

El ejemplo en cuestión constará de dos secciones “independientes”.

* La primera sección mostrará de forma permanente la información almacenada en las tablas **usr** y **phone** de la base de datos. Para mostrar dicha información se utilizará un objeto de la clase **QTableView** que estará **vinculado** con un modelo (el datasource de dicha tabla) que será un objeto de la clase **QSqlQueryModel**. Esta primera sección también tendrá un campo de edición que permitirá al usuario realizar operaciones del tipo **update, insert y delete** sobre cualquiera de las tablas mostradas.
* La segunda sección mostrará el resultado de una consulta específica: dado un nombre de usuario listar los teléfonos inscritos a su nombre. Para mostrar esta información se utilizarán objetos del mismo tipo que en la sección anterior.
* Se utilizará la barra de estado (**StatusBar**) de la ventana principal para visualizar mensajes de interés como errores o consultas bien realizadas.

Una vista de la interfaz de la aplicación se muestra a continuación:



Widgets de relevancia de la sección 1: **tableView** (izquierda)**, tableView\_2** (derecha)**, lineEdit y pushButton**.

Widgets de relevancia de la sección 2: **tableView\_3, lineEdit\_2 y pushButton\_2**.

Los cuadros rectangulares usados son **Frames**.

En el fichero .pro de todo proyecto de Qt que maneje bases de datos se debe agregar la línea: **QT += sql**



|  |
| --- |
| **Fichero mainwindow.h** |

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QtSql/QSqlQueryModel>

#include <QtSql/QSqlError>

QT\_BEGIN\_NAMESPACE

namespace **Ui** { class **MainWindow**; }

QT\_END\_NAMESPACE

class **MainWindow** : public QMainWindow

{

 Q\_OBJECT

public:

 **MainWindow**(QWidget \*parent = nullptr);

 ~***MainWindow***();

private slots:

 void **on\_pushButton\_clicked**();

 void **on\_pushButton\_2\_clicked**();

private:

 Ui::MainWindow \*ui;

 QSqlQueryModel usrTable;

 QSqlQueryModel phoneTable;

 QSqlQueryModel selectTable;

 QSqlQueryModel cTable; // para realizar las consultas sobre

 //cualquiera de las tablas

};

#endif // MAINWINDOW\_H

**Notas: QSqlQueryModel y QSqlError** son ficheros que incluyen la declaración de las clases del mismo nombre. La primera, utilizada como fuente de datos, vincula la base de datos con la vista correspondiente (**QViewTable**). En este fichero se definen dos slots privados para dar tratamiento a las señales **clicked()** de los dos botones **ejecutar** presentes en la interface de la aplicación.

Por último, se definen nuevos objetos miembros de la clase MainWindow: usrTable, phoneTable, selectTable y cTable, todos de la clase QSqlQueryModel. Estos objetos manejarán las consultas relacionadas con las vistas de la tabla usr (usrTable), phone (phoneTable), la consulta de la sección 1 (selectTable) y las consultas de la sección 2 (cTable).

|  |
| --- |
| **Fichero mainwindow.cpp** |

#include "mainwindow.h"

#include "ui\_mainwindow.h"

static bool **createConnection**()

{

 QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");

 db.setDatabaseName("C:/Users/alexa/Documents/bd\_visual/bd\_ej.db"); **//Modifique**

 if (!db.open()) return 0;

 else return 1;

}

MainWindow::**MainWindow**(QWidget \*parent)

 : QMainWindow(parent)

 , ui(new Ui::MainWindow)

{

 ui->setupUi(this);

 if (!createConnection()){

 statusBar()->showMessage("Error al abrir la base de datos."

 "Compruebe su localización.");

 return;

 }

 statusBar()->showMessage("Base de datos abierta :-)");

 //inicializar las clases que manejan el modelo de la tabla usr y phone.

 usrTable.setQuery("SELECT \* FROM usr;");

 usrTable.*setHeaderData*(0, Qt::Horizontal, "Usr ID");

 usrTable.*setHeaderData*(1, Qt::Horizontal, "Name");

 phoneTable.setQuery("SELECT \* FROM phone;");

 phoneTable.*setHeaderData*(0, Qt::Horizontal, "Phone ID");

 phoneTable.*setHeaderData*(1, Qt::Horizontal, "Usr ID");

 phoneTable.*setHeaderData*(2, Qt::Horizontal, "Phone No.");

 selectTable.setQuery(";");

 selectTable.*setHeaderData*(0, Qt::Horizontal, "Phone ID");

 selectTable.*setHeaderData*(1, Qt::Horizontal, "Usr Name");

 selectTable.*setHeaderData*(2, Qt::Horizontal, "Phone No.");

 ui->tableView->*setModel*(&usrTable);

 ui->tableView\_2->*setModel*(&phoneTable);

 ui->tableView\_3->*setModel*(&selectTable);

 ui->tableView->show();

 ui->tableView\_2->show();

 ui->tableView\_3->show();

}

MainWindow::~***MainWindow***()

{

 delete ui;

}

void MainWindow::**on\_pushButton\_clicked**()

{

 // la consulta se aplica sobre una de las dos tablas.

 // después se actualizan las dos.

 QString stmt = ui->lineEdit->text();

 cTable.setQuery(stmt); // ejecutamos la consulta

 if (cTable.lastError().type()==QSqlError::NoError) {

 usrTable.setQuery("SELECT \* FROM usr;"); // Actualizar las tablas.

 phoneTable.setQuery("SELECT \* FROM phone;");

 statusBar()->showMessage("Consulta realizada con éxito :-)");

 }

 else {

 // Si hubo error entonces se comunica

 statusBar()->showMessage("Error al realizar la última consulta :-(");

 }

 // limpiar el campo de edición de consultas

 ui->lineEdit->clear();

}

void MainWindow::**on\_pushButton\_2\_clicked**()

{

 QString usrname = ui->lineEdit\_2->text();

 QString stmt = "SELECT id\_phone,n\_usr,phone FROM usr,phone WHERE "

 "usr.id\_usr=phone.id\_usr AND usr.n\_usr=\"" + usrname + "\";";

 selectTable.setQuery(stmt);

 if (selectTable.lastError().type()==QSqlError::NoError) {

 statusBar()->showMessage("Consulta realizada con éxito :-)");

 }

 else {

 // Si hubo error entonces se comunica

 statusBar()->showMessage("Error al realizar la última consulta :-(");

 }

 // limpiar el campo de edición de consultas

 ui->lineEdit\_2->clear();

}

Recuerde modificar la dirección o camino de la base de datos a su propio directorio de trabajo de su proyecto.

**Notas:**

1. La función **static bool createConnection()**como ya se vio en el ejemplo de la conferencia establece la conexión con una base de datos relacional. En este caso se indica el tipo de base de datos en la primera sentencia (**QSQLITE** para una **BD SQLite**), se define un nombre para la base de datos, y se intenta abrir. La función devuelve 1 (true) si se pudo abrir con éxito y 0 (falso) si no se pudo hacer.

2. El método **MainWindow::MainWindow(QWidget \*parent) : QMainWindow(parent), ui(new Ui::MainWindow)** es el método constructor de la clase ventana **MainWindow**. En este método se realizan todas las inicializaciones necesarias.

* Se trata de abrir la conexión con la BD invocando a la función createConnection(). Si no se puede crear la conexión se visualiza un mensaje de error en la barra de estado de la ventana principal.
* Si se pudo abrir la conexión, se muestra un mensaje en la barra de estado de la ventana principal que indica el éxito de la operación, y se inicializan los objetos correspondientes a los modelos (fuentes de datos) de los objetos **QTableView** de la aplicación, que como se recuerda, son los que visualizan la información almacenada en las tablas de acuerdo a una consulta. Las 3 vistas (**QTableView**) tienen el mismo tipo de modelo por lo que se explicará uno de ellos:

**usrTable.setQuery("select \* from usr;");**

**usrTable.*setHeaderData*(0, Qt::Horizontal, "Usr ID");**

**usrTable.*setHeaderData*(1, Qt::Horizontal, "Name");**

La vista que permite visualizar la fuente de datos usrTable es un **QTableView**. Definiendo el modelo podemos definir la apariencia y elementos que contendrá la vista de tipo **QTableView (ver la figura que muestra la interfaz)**. En las 3 instrucciones anteriores se fija la consulta que determinará los elementos (registros) de la fuente de datos (usrTable), así como se definen dos columnas cabeceras de la tabla, con índices 0 y 1 y textos Usr ID y Name.

* Las últimas 6 instrucciones del constructor son respectivamente para asociar el modelo con cada una de las vistas (objetos tableView, tableView\_2 y tableView\_3) y a mostrarlas al invocar al método show().

3. Por último los slots **void MainWindow::on\_pushButton\_clicked() y void MainWindow :: on\_pushButton\_2\_clicked()** son muy parecidos y responden a las señales **clicked()** de los dos push buttons (con texto ejecutar) de la ventana principal. En el código se pueden apreciar comentarios que explican cada una de las sentencias por lo que se repasará a continuación la secuencia lógica de lo que hay que hacer:

* Leer la entrada de texto de la interface de la aplicación.
* Crear una cadena (de tipo **QString** o de otro tipo) con la consulta a realizarse. Si es necesario se debe componer la cadena a partir de varias cadenas hasta armar la consulta (ver **void MainWindow::on\_pushButton\_clicked()**).
* Fijar la sentencia SQL en el modelo. Esto ejecuta una consulta y actualiza los registros almacenados en el modelo como resultado de la consulta.
* Dar tratamiento de errores para el caso en que la sentencia no se haya podido ejecutar correctamente. Imprimir error en la barra de estado u operación exitosa.
* En el caso en que sea necesario, se debe actualizar el modelo. Este puede ser el caso de las vistas de la sección 2. Recuerde que las sentencias INSERT, DELETE o UPDATE no devuelven los registros de la tabla, por lo que si se quieren seguir visualizando se debe actualizar el modelo con una consulta SELECT. Este es el caso del método **void MainWindow::on\_pushButton\_2\_clicked()** y las sentencias

usrTable.setQuery("select \* from usr;");

phoneTable.setQuery("select \* from phone;");

Ejecute la aplicación y revise su correcta ejecución. Adicione sentencias SQL en la sección 1 para que pueda apreciar otros resultados.

**Estudio Independiente.**

En dependencia de la necesidad de su utilización en el proyecto, se recomienda revisar los ejemplos que tiene incorporados Qt sobre bases de datos. Puede tener acceso a ellos en la sección de bienvenida, buscando (donde dice **Choose and Example…**) la categoría SQL. Dichos ejemplos muestran algunas diferencias entre las clases que pueden servir de fuentes de datos (datasource) y las que sirven de vista, que conviene conocer.

**Nota:** Esta prepráctica no lleva informe, se entrega la carpeta del proyecto compactada.

**BIBLIOGRAFIA**

[1]. Zhi Eng, Lee; Rischpater, Ray. Application Development with Qt Creator: Build cross-platform applications and GUIs using Qt 5 and C++, 3rd Edition. (2020). Packt Publishing. ISBN-10: 1789951755, ISBN-13: 978-1789951752.

[2] [http://www.qtrac.eu/C++-GUI-Programming-with-Qt-4-1st-ed.zip](http://www.qtrac.eu/C%2B%2B-GUI-Programming-with-Qt-4-1st-ed.zip)

[3] Qt 5.14.2 Reference Documentation. Qt Creator Help.

[4] <https://doc.qt.io/qtcreator/index.html>

[5] <https://doc.qt.io/qt-5/qtdesigner-manual.html>

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| **INICIAL** | **EN DESARROLLO** | **DESARROLLADO** | **EXCELENTE** |

**Firma del Profesor**