© Kiskapu Kft. Minden jog fenntartva

Egységes és objektumközpontú adatbázis-kezelés

A Qt könyvtár adatbázis-kezelő alrendszerének bemutatása.

Néhány éve még irigykedve néztük a Windows (ADO, BDE) és Java (JDBC) környezetek által megvalósított adatbáziskezelő-független és objektumközpontú programalrendszereket. Természetes programozói igény, hogy a forráskód egyik adatkezelő környezetből könnyen átvihető legyen a másikba. Vajon miért? Lehetséges, hogy idővel nagyobb tudású adatbázis-kezelő környezetre lesz szükségünk, de az is gyakori, hogy a programot használók különféle adatbázismotorokat szeretnének használni. A nyílt ODBC szabvány régóta létezik Linuxra is, de ennek több komoly hátránya is akad, amelyek közül most csak kettőt említünk meg:

- Az ODBC nem objektumközpontú API.
- Az ODBC és a grafikus GUI-elemek adattartalmainak az illesztése nehézkes. Ez azt jelenti, hogy az adatforrások (táblák, nézetek) és a megjelenítésvezérlők együttműködése nem valósul meg önműködően egy adatforrásban szereplő összekötő objektumon keresztül.

Ez a helyzet mára már gyökeresen megváltozott. A Qt/KDE és Gtk+/Gnome rendszerek egyaránt fejlett, objektumközpontú API-modulokkal rendelkeznek. A Gnome környezet a GNOME-DB alrendszert bocsátja rendelkezésünkre, míg a Qt/KDE a Qt SQL-modult. Mindkét programozói környezet kiváló minőségű és teljes. Ebben a cikkben most a Qt/KDE SQL-környezetet szeretném bemutatni.

A Qt adatbázis-kezelő API felépítése

A Qt SQL-modul három programrétegből áll. A legalsó az illesztőréteg, ami az adatbáziskezelő-függő részeket tartalmazza, és egy felületet biztosít az adatbáziskezelő-független SQL API réteg számára. Az illesztőréteg jól meghatározott szabályok szerint épül fel, így egy új adatbázis-kezelő rendszer összevonása a Qt SQL-modulba könnyen megvalósítható. A legismertebb adatbázis-kiszolgálók illesztőit a Qt-csomag tartalmazza:

```
ORACLE (QOCI8)
PostgreSQL (QPSQL7)
MySQL (QMYSQL3)
MS SQL szerver és Sybase (QTDS7)
UNIX ODBC használata (QODBC3)
```

A fenti felsorolásban zárójelben adtuk meg azokat a karakterlánc-szimbólumokat, amelyek az adott illesztő indítását kérik. A SQL API réteg felépítése hasonló más hasonló API-k szerkezetéhez. Tekintsük át röviden az ebben a rétegben lévő adatelérő és -kezelő osztályokat:

- `QSqlDatabase` osztály: a kapcsolatosztályt (`session`, `connection`) valósítja meg. Ennek az osztálynak az objektumai teszik lehetővé például az adatbázishoz való kapcsolódást, illetve a tranzakció-kezelést.
- `QSqlQuery` osztály: egy tetszőleges SQL-parancs (`select`, `insert`, `update`, `delete`, `create...`, `alter...`) kiadását teszi lehetővé. Lekérdezés esetén az eredményhalmaz kezelését is támogatja.

táblázat1.exe

	Hónap	Nap	Névnap
1	1	1	Fruzsina
2	1	2	Ábel
3	1	3	Genovéva, Benjá
4	1	4	Titusz, Leona
5	1	5	Simon
6	1	6	Boldizsár
7	1	7	Attila, Ramóna
8	1	8	Gyöngyvér
9	1	9	Marcell
10	1	10	Melánia
11	1	11	Ágota
12	1	12	Ernő
13	1	13	Veronika

1. kép A táblázat1.exe program futási képe

- `QSqlCursor` osztály: a táblák, nézetek soraiból összeállított klasszikus SQL-kurzor kezelését valósítja meg. Az eredménytábla adataihoz való hozzáférés mellett az SQL-kurzoron keresztül történő adatmódosító műveleteket (`insert`, `update`, `delete`) is támogatja. Ennek az osztálynak van egy nagyon fontos további szerepe is, ugyanis adatforrásul működik a GUI-vezérlők számára.
- `QSqlRecord` osztály: egy tábla vagy nézet egy sorát képviseli, illetve a sornak megfelelő adatmezők gyűjteményét tartalmazza. Lehetővé teszi a sor adatainak (adatmezők) lekérdezését és módosítását.
- `QSqlField` osztály: egy tábla vagy nézet egy-egy adatoszlopának a kezelését tudja.
- `QSqlError` osztály: az SQL-műveletek során fellépő hibákról ad részletesebb tájékoztatást.
- `QSqlIndex` osztály: az adatbázisindexek és a kurzorok rendezettségét támogató osztály.

A Qt SQL alrendszer utolsó programrétege a felhasználói illesztőréteg, vagyis a grafikus elemek (data-aware widgets) – nagyrészt ezek végett használjuk a Qt könyvtárat. Itt jegyezzük meg, hogy a Qt SQL használatához nem szükséges ennek a szintnek az igénybevétele, azaz karakteralapú (`ncurses`, `tvision`) programokat is készíthetünk a Qt SQL API használatával. Ezek a vezérlők önműködően kezelik adatbázisunk adatait, ebben a szerkezetben a `QSqlCursor` osztálybeli objektum tölti be az adatforrás szerepét. Tekintsük át e réteg fontosabb osztályait:

- `QDataBrowser` osztály: ennek az osztálynak a használatával adatbevivő űrlapok készíthetők, illetve az adatbázisban böngésző űrlapok készítését is támogatja.
- `QDataTable` osztály: szerepe hasonló a `QDataBrowser` osztályéhoz, de a táblázatos megjelenítést támogatja.

© Kiskapu Kft. Minden jog fenntartva

- `QDataView` osztály: amennyiben csak olvasható űrlapot szeretnénk készíteni, úgy ebből az osztályból érdemes építkezni. Kinézete és működése a `QDataBrowser` osztályéhoz hasonló.
- `QSqlForm` osztály: ez az osztály teszi lehetővé és kezeli azt, hogy az általános Qt-vezérlőkből adatbázis-kezelő űrlapokat építhessünk.
- `QSqlPropertyMap` osztály: ennek az osztálynak egy-egy objektuma hordozza azt az adatot, amely megmondja, hogy a GUI-vezérlők mely adatbázistábla melyik mezőjéhez vannak rendelve. Ki szeretnénk emelni, hogy e cikk készítése során végig a PostgreSQL adatbázis-kezelőt használtuk.

2. kép A form1. exe

Hogyan készíthetünk Qt alapú adatbázis-kezelő programokat?

Legyen egy `cs_adatok` nevű PostgreSQL adatbázisunk, amiben egy névnapokat tartalmazó `nevek` nevű tábla szerepel. A tábla szerkezete egyszerű, három oszlopa van:

```
honap - number
nap - number
nevnap - varchar(50)
```

A feladatunk az, hogy szöveges üzemmódban maradva listázzuk ki a képernyőre a tábla tartalmát. Íme a feladatot megoldó C++-forráskód (*1. lista*, lásd még az 54. CD-n a Magazin/Qt/sql_1.cpp-t):

A 14. sorban az `app` objektumot csak létre kell hozni, semmilyen további teendő nincs vele. A `QApplication` osztály létrehozójának 3. kapcsolója azt vezérli, hogy grafikus vagy karakteralapú programot szeretnénk-e készíteni. Jelen esetben ennek a kapcsolónak `false` értéket adtunk, ami a szöveges alapú alkalmazások készítését teszi lehetővé. Ez a kapcsoló alapértelmezésben `true`, azaz ekkor a Qt grafikus összetevőit is használhatjuk. A 15. sorban lévő `pg` mutató egy adatbázis-kapcsolatot jelöl. Látható, hogy a `QSqlDatabase` osztály statikus `addDatabase()` tagfüggvényét hogyan láttuk el értékkel a `QPSQL7` karakterlánccal, azaz megmondjuk, hogy a `pg` mutatón keresztül egy PostgreSQL adatbázist szeretnénk majd kezelni, ami egy adatbáziskezelő-illesztőigénylést is jelent.
A 19–23. sor a kapcsolódáshoz szükséges bejelentkezési adatok kitöltését szemlélteti, amit természetesen a `pg` által mutatott objektum fog tartalmazni. Érdekességként megjegyezzük, hogy Oracle-adatbázis esetén a `setDatabaseName()` tagfüggvény értéke a `TNS` név. A kód 25. sorában megpróbálunk az adatbázishoz kapcsolódni. Az ügyfél-kiszolgálókapcsolat felépülése után a 27. sorban létrehozunk egy `q` SQL lekérdezést lehetővé tevő objektumot olyan módon, hogy a létrehozójában mindjárt meg is adjuk a végrehajtandó SQL-parancsot, amely még a létrehozóban végre is hajtódik. A 28. sor `isActive()` tagfüggvénye akkor lesz igaz, ha az SQL-lekérdezés eredménytáblája sikeresen létrejött.
A 30–35. sorban egy ciklusban kiírjuk a lekért tábla összes sorát a képernyőre. Az eredmény táblában a `next()` tagfüggvény áll a következő sorra, amit működő sornak is nevezünk. A `q` objektum `value()` tagfüggvénye teszi elérhetővé a lekért adatokat. A program befejezése előtt a 38. sorban lezárjuk az adatbázis-kapcsolatot.
Ennyi magyarázat után tekintsük át az ***sql_1.cpp*** program fordítását! Érdemes egy `Makefile`-t készíteni hozzá a következő tartalommal, amelyben a `-I` és `-L` utáni útvonal a Qt telepítési helyétől függően eltérő is lehet:

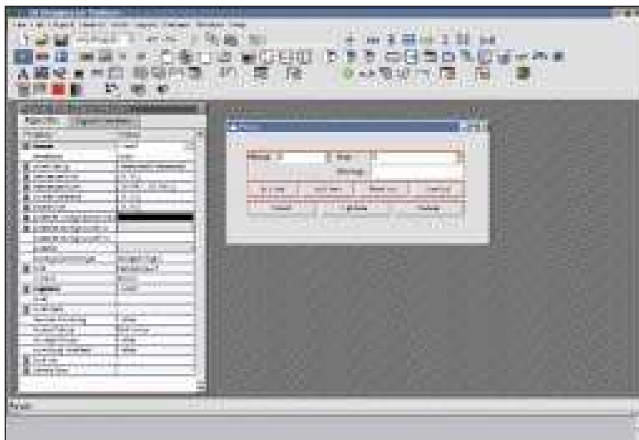
1. lista Az sql_1.cpp forráskódja

```
1. //
2. // sql_1.cpp
3. //
4. #include <iostream>
5. #include <qapplication.h>
6. #include <qsqldatabase.h>
7.
8. using std::cout;
9.
10.//--- A program indulási pontja ---
11.int main(int argc, char **argv)
12.{
13.
14. QApplication app(argc, argv, false);
15. QSqlDatabase *pg =
    ➥QSqlDatabase::addDatabase( "QPSQL7" );
16.
17. if ( !pg ) { cout << "DRIVER hiba!\n";
                              ➥return 1; }
18.
19. pg->setDatabaseName("cs_adatok");
20. pg->setUserName("postgres");
21. pg->setPassword("111111");
22. pg->setHostName("localhost");
23. pg->setPort( 5432 );
24.
25. if ( !pg->open() ) { cout << "Hiba az
    ➥AB nyitáskor!\n"; return 1; }
26.
27. QSqlQuery q("select * from nevek;");
28. if ( q.isActive() )
29. {
30. while ( q.next() )
31. {
32. cout << "\n" << q.value( 0 ).toString()
        ➥<< "\t";
33. cout << q.value( 1 ).toString() << "\t";
34. cout << q.value( 2 ).toString();
35. }
36. }
37.
38. pg->close();
39.
40. return 0;
41.}*
```

```
CXFLAGS = -I/usr/lib/qt3/include
➥-L/usr/lib/qt3/lib
LIBS = -lqt-mt
sql_1.exe : sql_1.cpp
g++ $(CXFLAGS) -o sql_1.exe $(LIBS) sql_1.cpp
```

Ezek után adjuk ki a `make` parancsot, ami elkészíti az ***sql_1.exe*** bináris programot, amit ha végrehajtunk, láthatjuk táblánk végiggördülő tartalmát a képernyőn.
Megjegyzés: programunkban természetesen több `QSqlDatabase` objektum is lehet, hiszen néha olyan a feladatunk, hogy több – akár különböző típusú – adatbázisból kell

© Kiskapu Kft. Minden jog fenntartva



3. kép A QT Designer

2. lista Általános SQL-parancsok

```
q.exec("create table ujnevek as select
➥* from nevek;");
q.exec("alter table ujnevek add constraint
➥pk_ujnevek primary key(honap,
nap);");
q.exec("select * from ujnevek;");
q.exec("insert into ujnevek (honap, nap,
➥nevnap) values ('1', '1', 'Linus')" );
```

3. lista Tranzakció-kezelés

```
1. if ( pg->transaction() )
   // Egy tranzakciót kezdünk, ha tudunk
2. {
3. q.exec("insert into ujnevek (honap, nap,
   ➥nevnap) values ('1', '1', 'Linus')" );
4. pg->commit();
5. }
6. else
7. {
8. cout << "\nA tranzakció megkezdése
   ➥nem sikerült!";
9. }
```

egyszerre dolgoznunk. Ilyenkor az adatbázis-kezelő műveleteket megvalósító tagfüggvények egyik értékeként mindig jeleznünk kell, hogy melyik munkamenetre irányul a kérés.

Az általános SQL parancskezelő osztály

A `QSqlQuery` osztály egy általános SQL parancsfogadó szolgáltatást valósít meg. Amennyiben a `pg` mutató egy érvényes adatbázis-kapcsolatot ír le, akkor a következő változómeghatározást is használhatjuk:

```
QSqlQuery q("", pg); // A q a közvetlen SQL
parancsok végrehajtója
```

Látható, hogy a létrejött `q` objektum a `pg` által megvalósított adatbázisra vonatkozóan fog tudni SQL-parancsokat elküldeni. A létrehozó első értékében azt is jeleztük, hogy nem kérünk azonnali parancsvégrehajtást. A `q.exec()` tagfüggvény szolgál a parancsok végrehajtására, ezek bármilyen érvényes SQL-utasítások lehetnek, (lásd például a *2. listát*).
A `q` objektum a `select` lefutása után magát az SQL-kurzort, azaz az eredménytáblát és annak adatelérését is tartalmazza. Ennek megfelelően léteznek például az eredménytábla sorain mozgó `first(),next(),last()...` tagfüggvények. Az `at()` tagfüggvény visszaadja, hogy az aktív sor hányadik (az indexelés 0-tól kezdődik). A `seek()` tagfüggvény még a közvetlen pozicionálást is lehetővé teszi. A `size()` az eredménysorok számát adja vissza. Megjegyezzük, hogy a C nyelv `sprintf()` függvénye segítségével könnyen létre tudunk értékekkel megtűzdelt SQL-parancsokat hozni, például:

```
char sqltxt[200];
sprintf(sqltxt, "select * from nevek where
➥honap=%d and nap=%d", 11, 5);
```

Tranzakció-kezelés

A tranzakció-kezelés természetesen a `QSqlDatabase` osztály (a munkamenet, azaz session) objektumán keresztül valósul meg. Legyen a `pg` egy ilyen osztály objektumára mutató változó. Ekkor a tranzakciót kezelő kód jellemzően ilyen szerkezetű lesz (*3. lista*).
Az 1. kódsor `pg->transaction()` tagfüggvénye kijelöli a tranzakció kezdetét. Lehetséges, hogy az általunk használt adatbázis-kezelő nem tud tranzakciókat kezelni, ezt az esetet a 6–9. sorok kezelik le.
A 3. sor `insert` művelete után a `pg->commit()` és a `pg->rollback()` hívást is választhatnánk, azaz véglegesíthetjük a változtatásokat, illetve el is vethetjük őket. Esetünkben a 4. sorban egy `commit()` véglegesíti az új rekord beszúrását.
A tranzakció-kezelést az 54. CD Magazin/Qt/sql_2.cpp program használatával próbálhatjuk ki.
Sorozatunk következő részében a Qt alapú SQL kurzorok használatát, a hibakezelést, valamint az adatbázis-kezelő GUI készítési Qt lehetőségeket fogjuk áttekinteni. Szó lesz az űrlap (form) alapú alkalmazások készítésének módszeréről, de röviden kitérünk a Qt Designer használatára is.

Nyíri Imre (inyiri@mol.hu)
Jelenleg a MOL Rt.-nél dolgozik. Informatikai vállalkozásában az Internet, a Linux, valamint a Java-programozás gyakorlati hasznosításával foglalkozik. Örök szerelme a C++ maradt.