Fundusze Europejskie
Wiedza Edukacja Rozwój

Unia Europejska
Europejski Fundusz Społeczny

Programowanie zespołowe 2.0

Tworzenie aplikacji bazodanowych z wykorzystaniem biblioteki Qt

SCENARIUSZ ZAJĘĆ

Czas realizacji: 180 minut
Cele ogólne: Przedstawienie podstawowych klas biblioteki Qt przeznaczonych do pracy z relacyjnymi bazami danych.
Cel szczegółowy: Przygotowanie prostej aplikacji, która łączy się z istniejącą bazą danych oraz pozwala na pobieranie i edycje danych w tej bazie.
Konieczne wiadomości wstępne: Umiejętność programowania w języku C++ oraz bibliotece Qt. Podstawowa znajomość języka SQL oraz zagadnień związanych z relacyjnymi bazami danych.
Metoda prowadzenia zajęć: Dyskusja z elementami wykładu.

Pobieranie i edycja informacji znajdujących się w relacyjnej bazie danych bezpośrednio za pomocą wysyłanych do serwera zapytań jest niezbyt wygodne i wymaga od użytkownika znajomości języka SQL. Nie jest to najlepsze rozwiązanie również z punktu widzenia bezpieczeństwa danych. Znacznie lepszym pomysłem jest zaprojektowanie odpowiedniej aplikacji, która umożliwi wyświetlanie danych i wykonywanie na nich operacji za pomocą odpowiednich elementów interfejsu graficznego. Pozwoli również ograniczyć dostęp użytkownika tylko do wybranego zestawu poleceń operujących na danych. Biblioteka Qt pozwala na bardzo wygodne tworzenie aplikacji tego typu.

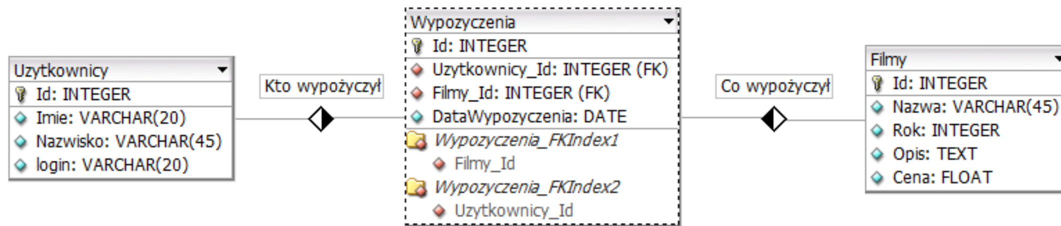
Baza danych

Celem niniejszych zajęć jest zbudowanie prostej aplikacji, która połączy się z bazą danych (umieszczoną na komputerze lokalnym lub zdalnym serwerze) i pozwoli użytkownikowi na wyświetlanie informacji w niej umieszczonych, a także na dodawanie, usuwanie i edycję danych.

Dalej będziemy zakładać, że na serwerze została utworzona prosta baza danych o strukturze takiej jak na rysunku 1. Założymy również, że baza została wypełniona przykładowymi danymi. Odpowiednie polecenia języka SQL zostały dołączone do niniejszego scenariusza. Metody tworzenia bazy i dodawania do niej danych zostały szczegółowo omówione w scenariuszu „Modelowanie i tworzenie baz danych".

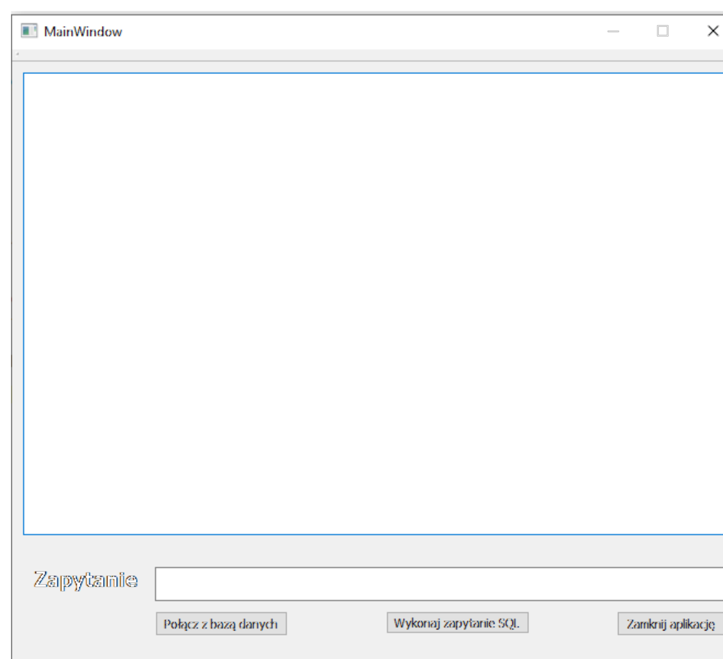
Łączenie się z bazą danych

Uruchamiamy program Qt Creator i otwieramy nowy projekt typu „Qt Widgets Application" (więcej informacji o tworzeniu aplikacji w Qt można znaleźć w poprzednim scenariuszu). Dalej będziemy zakładać, że projekt ma nazwę *baza1*. Następnie korzystając



Grafika 1: Diagram encji dla bazy internetowej wypożyczalni wideo

z *QtDesignera* tworzymy interfejs naszej aplikacji. Interfejs składa się z trzech przycisków, pola edycyjnego (`QLineEdit`) do wpisywania zapytań SQL oraz pola tekstowego typu `QTextEdit` do wyświetlania wyników. Elementy powinny zostać rozmieszczone tak jak na rysunku 2.



Grafika 2: Interfejs aplikacji

Aby móc korzystać z klas obsługujących bazy danych, musimy dodać do pliku naszego projektu moduł `sql` W tym celu do pliku *baza1.pro* dodajemy następujący wiersz

```
qt += sql
```

Teraz możemy utworzyć połączenie z bazą danych. Służy do tego klasa `QSqlDatabase`. Aby jej użyć w pliku *baza1.h* dodajemy odpowiedni plik nagłówkowy i prywatne pole tej klasy:

```
#include <QSqlDatabase>
...
private:
    QSqlDatabase baza;
```

Łączenie z bazą będzie następować po naciśnięciu odpowiedniego przycisku, tworzymy więc nowy slot połączony ze zdarzeniem `clicked()`. W wyniku otrzymujemy szkielet funkcji, w którym umieścimy za chwilę polecenia tworzące i otwierające połączenie z bazą:

```
void MainWindow::on_pushButton_2_clicked()
{

}
```

Klasa `QSqlDatabase` umożliwia połączenie się z popularnymi systemami baz danych m.in. *MySql*, *SQLite*, *PostgreSQL*, *IBM DB2*, *Microsoft SQL Server*. Każda z tych baz wymaga innego sterownika, którego nazwę podajemy tworząc instancję tej klasy za pomocą statycznej metody `addDatabase`. W naszym przypadku użyjemy sterownika `QPSQL` przeznaczonego do współpracy z bazami typu *PostgreSQL* w wersjach 7.3 i wyższej.

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

Następnie musimy określić najważniejsze parametry połączenia: nazwę hosta lub adres ip hosta, na którym postawiona została baza, nazwę bazy danych oraz login i hasło użytkownika bazy.

```
baza.setHostName("localhost");
baza.setDatabaseName("vod");
baza.setUserName("postgres");
baza.setPassword("haslo123");
```

W naszym przypadku baza o nazwie `vod` została umieszczona na komputerze lokalnym, a dostęp do niej uzyskamy korzystając z domyślnego konta użytkownika `postgres`. Oczywiście zamiast wklejać login i hasło do kodu źródłowego bezpieczniej byłoby utworzyć odpowiedni formularz logowania i prosić użytkownika o ich podanie przy każdym uruchomieniu aplikacji.

Obiekt reprezentujący połączenie z bazą został utworzony, ale fizyczne połączenie powstanie dopiero w momencie wywołania na nim metody `open()`.

```
bool czy_polaczono = baza.open()
```

jeżeli uda się otworzyć połączenie metoda zwraca wartość `true`, a w przeciwnym razie `false`. Zalecane jest sprawdzenie wyniku przed próbą wykonania kolejnych operacji związanych z bazą.

```
if (!czy_polaczono) {
    qDebug()<<"Błąd połączenia z bazą danych!";
}
else {
    qDebug()<<"Połączono z bazą";
    operacje bazodanowe
}
```

Ćwiczenie 1. Zapoznaj się z dokumentacją klasy `QSqlDatabase`. Zmień aplikację tak, aby to użytkownik podawał login i hasło do bazy danych. Wykorzystaj w tym celu odpowiednie elementy GUI.

Ćwiczenie 2. Metoda `LastError()` umożliwia uzyskanie bardziej szczegółowych informacji o błędach związanych operacjami bazodanowymi. Zapoznaj się z dokumentacją tej metody i dodaj ją do swojej aplikacji. Przetestuj jej działanie podając błędne parametry dostępu do bazy.

Ćwiczenie 3. Zmień metodę wyświetlania komunikatów o stanie połączenia. Zamiast konsoli `qDebug` wykorzystaj odpowiedni element GUI.

Ćwiczenie 4. Dodaj do aplikacji obsługę przycisku *Zamknij aplikację*. jego naciśnięcie powinno spowodować rozłączenie aplikacji z bazą danych (metoda `close()`) i zakończenie jej pracy.

Tworzenie i wykonywanie zapytań

Najprostszą metodą wykonania zapytania jest utworzenie odpowiedniego obiektu klasy `QSqlQuery` i wywołanie na nim metody `exec`

```
QSqlQuery zapytanie;
bool sukces = zapytanie.exec("SELECT Imie, Nazwisko FROM Uzytkownicy");
```

W przypadku zapytania typu `SELECT`, jeżeli zakończy się ono sukcesem (metoda `exec()` zwróci wartość `true`), to wyniki zapytania (zwrócone rekordy) możemy przeglądać korzystając z metod: `first()`, `last()`, `next()`, `previous()`, `seek()`. Do zawartości poszczególnych pól dostajemy się za pomocą metody `value(int indeks)`, gdzie indeks jest pozycją pola w zapytaniu `SELECT` (numeracja zaczyna się od 0). Typowy scenariusz wyświetlania wszystkich wyników zapytania wygląda następująco.

```
while(zapytanie.next()){
qDebug()<<"Imie: "<<zapytanie.value(0).toString();
qDebug()<<"Nazwisko: "<<zapytanie.value(1).toString();
}
```

Metoda `next()` zwraca `false` po dojściu do ostatniego rekordu zwróconego przez zapytanie. Metoda `value()` zwraca wartość ogólnego typu `QVariant`(tak naprawdę jest to tzw. *unia*), który należy odpowiednio zrzutować metodami typu `toString`, `toInt()`, itp.

Czasem (na przykład jeżeli chcemy wstawić do bazy jednocześnie wiele rekordów) korzystne jest oddzielenie treści zapytania od jego konkretnych parametrów. W tym celu

korzystamy z metody `prepare()`, która tworzy zapytanie, ale go nie wykonuje, a w treści zapytania wstawiamy tzw. *placeholdery*, które zastępujemy potem konkretnymi wartościami.

```
zapytanie.prepare("INSERT INTO Uzytkownicy(Id, Imie, Nazwisko, Login) "
                  "VALUES(:id, :imie, :nazwisko, :login)");
zapytanie.bindValue(:id,10);
zapytanie.bindValue(:imie,"Jan");
zapytanie.bindValue(:nazwisko,"Kowalski");
zapytanie.bindValue(:login,"janek");
zapytanie.exec();
}
```

Przy wstawianiu wielu rekordów metodę `prepare()` wywołujemy tylko raz. Wielokrotnie wywołujemy tylko metody `bindValue()` i `exec()`. W większości przypadków jest to sposób bardziej wydajny niż wielokrotne wywoływanie metody `exec()` z pełną treścią zapytania.

Ćwiczenie 5. Przerób program tak aby użytkownik mógł wprowadzać treść zapytań w przeznaczonym do tego polu tekstowym. Zapytanie powinno się wykonać po naciśnięciu przycisku *Wykonaj zapytanie SQL*.

Ćwiczenie 6. Spróbuj wykonać kilka innych zapytań. Sprawdź co dzieje się w przypadku gdy zapytanie zwróci pusty zbiór wyników.

Ćwiczenie 7. Zapoznaj się z dokumentacją klasy `QTextEdit`. Przerób program tak, aby wyniki zapytań zamiast na konsoli systemowej pojawiały się w polu tekstowym.

Klasa `QSqlQueryModel`

Biblioteka Qt zawiera również wysokopoziomowe klasy, które pozwalają na dostęp do danych w wygodniejszy sposób niż ten oferowany przez klasę `QSqlQuery`. Klasy te można łatwo zintegrować z odpowiednimi elementami interfejsu graficznego i umożliwiają tworzenie aplikacji zgodnych z wzorcem projektowym *Model-Widok-Kontroler*.

Pierwszą klasa tego typu jest `QSqlQueryModel`. Obsługuje ona dane w trybie tylko do odczytu, nadaje się więc doskonale do obsługi zapytań typu `SELECT`.

Obiektów tej klasy możemy też używać w podobny sposób jak zapytań typu `SqlQuery`.

```
QSqlQueryModel zapytanie;
    zapytanie.setQuery("SELECT id, nazwisko FROM Uzytkownicy");
    for (int i = 0; i < zapytanie.rowCount(); i++) {
        int id = zapytanie.record(i).value("id").toInt();
        QString nazwisko = zapytanie.record(i).value("nazwisko").toString();
        qDebug() << id << nazwisko;
    }
```

Metoda `rowCount()` podaje liczbę zwróconych rekordów. a metoda `record(int i)` pozwala uzyskać dostęp do i-tego ze zwróconych rekordów.

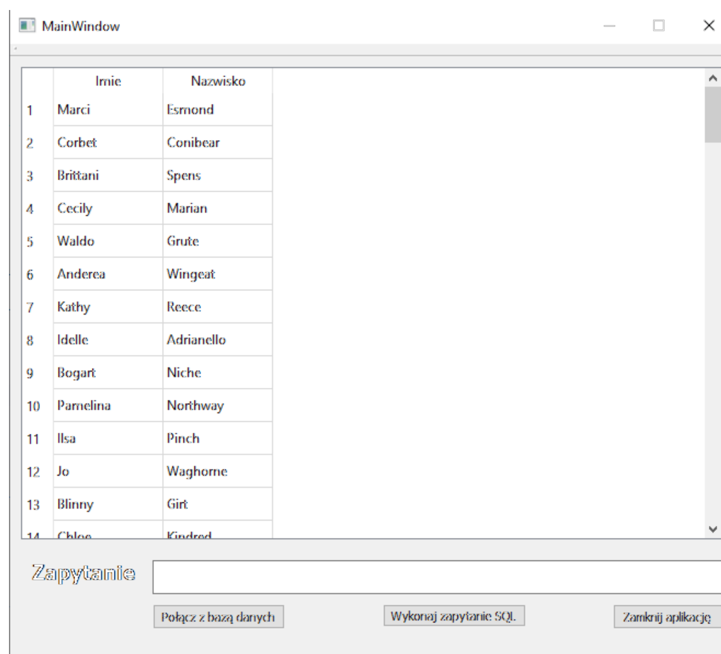
Zwykle jednak do wyświetlania wyników wykorzystujemy odpowiednią klasę GUI, np. `QTableView`. Klasa ta pozwala na wyświetlanie danych w postaci standardowej tabeli. Obiekty tej klasy możemy tworzyć programistycznie poleceniem

```
QTableView *widok = new QTableView;
```

lub za pomocą *QtDesignera*. Skorzystamy z tej drugiej metody i przeciągamy do okna naszej aplikacji widget *Table View)* (znajduje się w sekcji *Item Views (model based)*. [1] Następnie, za pomocą metody `setModel` musimy określić, że źródłem danych dla tabeli ma być nasze zapytanie i wywołać metodę `show()` aby wyświetlić wyniki.

```
QSqlQueryModel *zapytanie = new QSqlQueryModel;
zapytanie->setQuery("SELECT imie, nazwisko FROM Uzytkownicy");
zapytanie->setHeaderData(0, Qt::Horizontal, "Imię");
zapytanie->setHeaderData(1, Qt::Horizontal, "Nazwisko");
ui->tableView->setModel(zapytanie);
ui->tableView->show();
```

Metoda `setHeaderData` służy do ustawiania nagłówków poszczególnych kolumn (pól zwracanych przez zapytanie). Po wykonaniu powyższego kodu powinniśmy zobaczyć widok podobny do tego na rysunku 3.



Grafika 3: Wyświetlanie wyników zapytania za pomocą klasy `QTableView`

Ćwiczenie 8. Zapoznaj się z dokumentacją klasy `QTableView` oraz właściwościami tego obiektu w *QtDesignerze*. Zmodyfikuj program tak, aby nie były wyświetlane numery wierszy, a szerokość kolumn dopasowała się do szerokości tabeli.

[1]Wcześniej warto usunąć lub ukryć duże pole tekstowe, w którym dotychczas wyświetlaliśmy wyniki.

Ćwiczenie 9. Przerób program tak aby użytkownik mógł wprowadzać treść zapytań w przeznaczonym do tego polu tekstowym. Zapytanie powinno się wykonać po naciśnięciu przycisku *Wykonaj zapytanie SQL*. Dodaj przycisk czyszczący wyniki zapytania.

Klasa `QSqlTableModel`

Jeżeli chcemy modyfikować dane w tabelach możemy wykorzystać klasę `QSqlTableModel`. Umożliwia ona odczyt i zapis danych z pojedynczej tabeli. Korzystamy z niej w podobny sposób jak z klasy `QSqlQueryModel`. Metoda `setTable` pozwala wybrać tabelę, z której będą pobierane dane, a metoda `select` powoduje pobranie danych z bazy i przekazanie ich do obiektu `QSqlTableModel`.

```
QSqlTableModel *model = new QSqlTableModel();
model->setTable("Uzytkownicy");
model->setHeaderData(1, Qt::Horizontal, "Imię");
model->setHeaderData(2, Qt::Horizontal, "Nazwisko");
model->setHeaderData(3, Qt::Horizontal, "Login");
model->setEditStrategy(QSqlTableModel::OnFieldChange);
model->select();

ui->tableView->setModel(model);
ui->tableView->hideColumn(0);
ui->tableView->show();
```

Po wywołaniu powyższego kodu ujrzymy widok podobny do tego z rysunku 3 (widoczna będzie dodatkowa kolumna *login*). Zwróćmy uwagę, że kolumna zawierająca id użytkowników została ukryta za pomocą odpowiedniej metody klasy `QTableView`.

Jeżeli klikniemy dwukrotnie w pole tabeli, będziemy mogli zmienić jego zawartość. Metoda `setEditStrategy` określa kiedy zmiany dokonane przez użytkownika zostaną zapisane w bazie. Parametr `OnFieldChange` okresla, że zmiany zostaną zapisane w bazie bezpośrednio po zmianie pola tabeli. Inne opcje to `OnRowChange` – zmiany zapiszą się, kiedy użytkownik wybierze inny wiersz tabeli i `OnManualSubmit` – zmiany zapiszą się po wywołaniu metody `submitAll()`.

Jak dodawać i usuwać rekordy? Służą do tego metody: `insertRecord` – wstawia rekord do bazy, `insertRows` – wstawia puste rekordy do bazy, `removeRows` – usuwa rekordy z bazy. Aby je wykorzystać należy je połączyć z odpowiednimi zdarzeniami obiektu `QTableView` lub innych elementów GUI. Można np. dodać do aplikacji przycisk wstawiający do tabeli nowy pusty wiersz, a potem wpisać do niego odpowiednie wartości. Można dodać inny przycisk, który usunie z tabeli zaznaczone wiersze, lub dodać menu kontekstowe z opcją usuwania wierszy.

Ćwiczenie 10. Zaimplementuj funkcjonalności dodawania i usuwania wierszy.

Ćwiczenie 11. Zapoznaj się z metodami `setFilter` oraz `setSort`, a następnie spróbuj wykorzystać je w aplikacji.

Ćwiczenie 12. (Trudne!) Jeżeli pole tabeli zawiera tylko kilka unikalnych wartości (np. pole *wykształcenie* może mieć wartość *podstawowe*, *średnie* i *wyższe*), to zamiast wpisywać te wielokrotnie te wartości, wygodniej byłoby wybierać je z rozwijalnej listy (tzw. *ComboBoxa*). Dodaj taka możliwość do aplikacji.

Ćwiczenie 13. Zastosuj metody opisane w tym rozdziale do wyświetlenia zawartości tabeli *Wypozyczenia*.

Klasa `QSqlRelationalTableModel`

Nasza baza zawiera tabelę *Wypozyczenia*, zawierającą informacje o tym kto wypożyczył jaki film. Możemy wyświetlić i edytować jej zawartość podobnie jak zrobiliśmy to poprzednio. Zamiast nazwisk użytkowników i tytułów filmów zobaczymy jednak tylko ich identyfikatory. Są to klucze obce do tabel *Uzytkownicy* i *Filmy*, w których znajdują się interesujące nas informacje. Niestety nie możemy ich wyświetlić, gdyż klasa `QSqlTableModel` pozwala pracować tylko z jedna tabelą. Ograniczenia tego nie ma klasa `QSqlRelationalTableModel`. Działa ona bardzo podobnie do swej poprzedniczki, ale umożliwia dodatkowo wygodna pracę z tabelami, której kolumny sa kluczami obcymi do innych tabel w bazie.

```
QSqlRelationalTableModel* model = new QSqlRelationalTableModel();
model->setTable("wypozyczenia");
model->setJoinMode(QSqlRelationalTableModel::LeftJoin);
model->setRelation(1,QSqlRelation("uzytkownicy","iduzytkownika","nazwisko"));
model->setRelation(2,QSqlRelation("filmy","idfilmu","tytul"));
model->setEditStrategy(QSqlTableModel::OnFieldChange);
model->setSort(0, Qt::AscendingOrder);
model->select();
model->setHeaderData(1, Qt::Horizontal, "Wypożyczający");
model->setHeaderData(2, Qt::Horizontal, "Tytuł filmu");
model->setHeaderData(3, Qt::Horizontal, "Data wypożyczenia");
```

Większość metod w powyższym kodzie jest nam znana z lektury poprzedniego rozdziału. Najważniejszy nowy element to metoda `setRelation`. Pozwala ona ustanowić relację pomiedzy tabelami. Na przykład wywołanie polecenia:

```
model->setRelation(1,QSqlRelation("uzytkownicy","iduzytkownika","nazwisko"));
```

oznacza, że 1. kolumna (numerujemy od 0) tabeli *wypozyczenia* jest kluczem obcym powiązanym z polem *iduzytkownika* z tabeli *uzytkownicy* oraz, że zamiast wartości klucza ma zostać wyświetlona zawartość odpowiedniego pola *nazwisko*. Metoda `setJoinMode` określa tryb łączenia tabel (więcej informacji w scenariuszu poświęconym językowi SQL), a metoda `setSort` kryterium sortowania danych.

Utworzenie modelu to dopiero pierwszy krok, musimy go jeszcze połączyć z odpowiednim elementem GUI:

```
ui->tableView->setModel(przydzialy);
ui->tableView->setColumnHidden(0,true);
ui->tableView->setItemDelegate(new QSqlRelationalDelegate(ui->tableView));
ui->tableView->show();
```

Dzięki metodzie `setItemDelegate` będziemy mogli ustawiać zawartość pól za pomocą wygodnych *ComboBoxów*.

Odnośniki i załączniki

[1] Dokumentacja biblioteki Qt: https://doc.qt.io/qt-5/classes.html

[2] Tutorial SQL Programming: https://doc.qt.io/qt-5/sql-programming.html

[3] J. Ganczarski, M. Owczarek, *C++. Wykorzystaj potęgę aplikacji graficznych*, Helion 2012.