Kurs języka Object/Delphi Pascal

na bazie implementacji Free Pascal.

AUTOR ŁUKASZ STAFINIAK

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

Jeśli zauważysz błędy na slajdach, proszę daj znać!

Wykład 10: Bazy danych

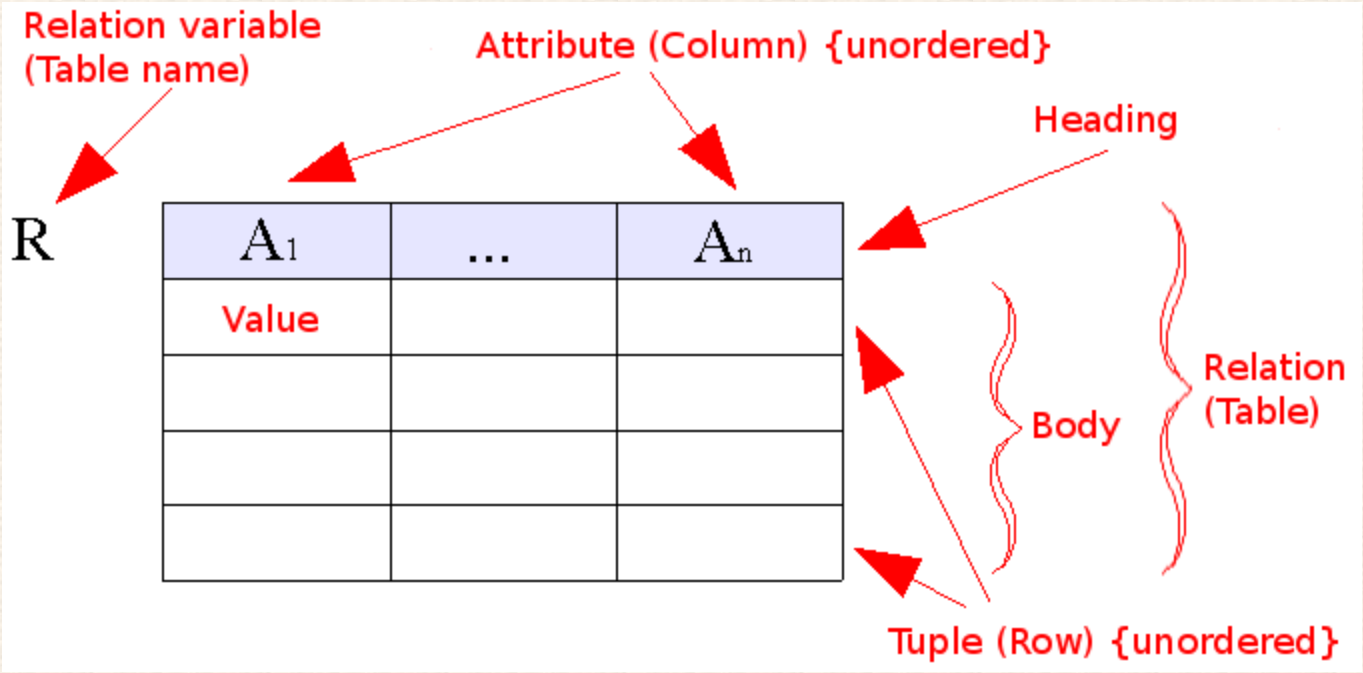
Łączenie z bazą danych.

Wyświetlanie danych tabelarycznych.

Relacyjne bazy danych

- Relacyjne bazy danych ciągle są najpopularniejszym schematem przechowywania ustrukturowanej informacji.
- Taka baza składa się z nazwanych **tabel**, które są zbiorami **krotek** (wierszy). Krotki są zbiorami przypisań wartości **atrybutom** (kolumnom). Wszystkie krotki tabeli muszą mieć ten sam zbiór atrybutów.

  en.wikipedia.org/wiki/Relational_model



- Atrybuty są otypowane, tabele wyglądają więc jak tablice rekordów w Pascalu (`array of record`) ale kolejność w tablicy się nie liczy.
- Dane z bazy pobieramy przy pomocy **zapytań** (*queries*), rezultat zapytania jest tabelą.
- **SQL** jest najpopularniejszym modelem / językiem relacyjnych baz danych
  - odbiega od czystego modelu relacyjnego, m.in.:
    - kolejność wierszy i kolumn się liczy
    - oraz mamy specjalną wartość `NULL` na brakujące pola krotek.
- Częścią bazy danych są **więzy** (*constraints*) zapewniające poprawność danych.
- Inne nowoczesne modele baz danych to **obiektowe bazy danych** (zgrubsza wzorowane na programowaniu zorientowanym obiektowo) oraz grafowe bazy danych, m.in. **bazy RDFowe** *triplet stores* przechowujące grafy skierowane z etykietowanymi krawędziami jako trójki obiekt-relacja-obiekt.
  - *Resource Description Framework* (RDF) dostarcza całej ontologii, tzn. uniwersalnej hierarchi klas i relacji-właściwości.

Polecenia SQLa

SQL jest bogatym językiem programowania, zdania to kwerendy/zapytania (*queries*) do bazy oraz instrukcje (*statements*) modyfikujące bazę danych.

en.wikipedia.org/wiki/SQL

`SELECT`. Kwerenda. Np:

```
SELECT isbn,                          Wyrażenia-atrybuty takie same w tabeli
       title,                         źródłowej (FROM) i w tabeli wynikowej.
       price,                         Złożone wyrażenie jako (AS) nowy atrybut.
       price * 0.06 AS sales_tax
  FROM Book                           Tabela źródłowa.
  WHERE price > 100.00                Warunek filtrujący.
  ORDER BY title;                     Porządek wierszy w tabeli wynikowej.
```

Używając `JOIN` łączymy atrybuty z kilku tabel:

```
SELECT *                    Wybierz wszystkie atrybuty z przeglądanych tabel.
FROM employee               Warunek równoważny z USING (DepartmentID);
INNER JOIN department ON employee.DepartmentID =
department.DepartmentID;
```

`INSERT`. Wprowadź do bazy. Np:

```
INSERT INTO My_table
        (field1, field2, field3)
    VALUES
        ('test', 'N', NULL);
```

Ilość atrybutów tutaj i wartości poniżej musi się zgadzać, ale brakujące kolumny przyjmą wartości domyślne

Można pominąć atrybuty jeśli używa się wszystkich, a też włożyć kilka krotek naraz:

```
INSERT INTO phone_book VALUES ('John Doe', '555-1212'),
                              ('Peter Doe', '555-2323');
```

`UPDATE`. Modyfikuj wiersze. Np:

```
UPDATE My_table
    SET field1 = 'updated value'
    WHERE field2 = 'N';
```

`DELETE`. Usuń istniejące wiersze. Np:

```
DELETE FROM My_table
    WHERE field2 = 'N';
```

`CREATE`. Stwórz nową tabelę. Np:

```
CREATE TABLE My_table(
    my_field1   INT,
    my_field2   VARCHAR(50),
    my_field3   DATE          NOT NULL,
    PRIMARY KEY (my_field1, my_field2)
);
```

`TRUNCATE`. Bezpowrotnie usuń wszystkie dane z tabeli, ale nie samą tabelę.

```
TRUNCATE TABLE My_table;
```

`DROP`. Usuń całą tabelę.

```
DROP TABLE My_table;
```

`COMMIT`. Często interakcja z bazą danych oparta jest o transakcje. `COMMIT` zatwierdza operacje dokonane od ostatniego wywołania `COMMIT`, inaczej nie są one widoczne dla innych uczestników i będą stracone jeśli zamkniemy klienta bez wywołania `COMMIT`.

Klucze obce

- Każda tabela ma **klucze** (kandydujące, *candidate keys*): zbiory atrybutów których wartości łącznie jednoznacznie identyfikują wiersz tabeli.
- Zbiór atrybutów może być **kluczem obcym** (*foreign key*) gdy jest kluczem innej tabeli – jego wartość jednoznacznie identyfikuje wiersz innej tabeli.
- Klucz obcy przy tworzeniu tabeli w SQL (en.wikipedia.org/wiki/Foreign_key):

```
CREATE TABLE TABLE_NAME (
   id    INTEGER  PRIMARY KEY,
   col2  CHARACTER VARYING(20),
   col3  INTEGER,
   ...
   FOREIGN KEY(col3)
      REFERENCES other_table(key_col) ON DELETE CASCADE,
   ... )
```

lub gdy klucz obcy jest pojedynczym atrybutem

```
CREATE TABLE TABLE_NAME (
   id    INTEGER  PRIMARY KEY,
   col2  CHARACTER VARYING(20),
   col3  INTEGER REFERENCES other_table(column_name),
   ... )
```

- ***Referential integrity*** to zapewnienie że nie ma „wiszących wskaźników", akcje `ON DELETE` / `ON UPDATE` na docelowym wierszu mogą wywołać:
  - `CASCADE`: gdy wiersze w tabeli docelowej są kasowane/modyfikowane, wiersze w danej tabeli też są kasowane/modyfikowany,
  - `RESTRICT`: blokada kasowania wierszy z odniesieniami,
    - `NO ACTION` jest łagodniejszą formą ale zgrubsza to samo,
  - `SET NULL`: wartości klucza obcego są ustawiane na `NULL` gdy wiersz docelowy jest kasowany/modyfikowany,
  - `SET DEFAULT`: kolumny (tzn. atrybuty) mają wartość domyślną, ustaw tą wartość gdy wiersz docelowy jest kasowany/modyfikowany.

Zagadnienia: plan

- Free Pascal posiada jako część *Free Component Library* jednolite API do obsługi różnych baz, a nawet różnych rodzajów baz danych (moduł `db`).
- Lazarus posiada komponenty pozwalające skonfigurować interakcję z bazą danych bezpośrednio z GUI Lazarusa.
- Duże bazy działają na zasadzie klient-serwer, gdzie bazą danych zarządza serwer, z którym komunikują się klienci wysyłając kwerendy czy instrukcje modyfikacji.
  - Tutaj Pascal rozmawia z „natywnym" klientem danej bazy, który dopiero komunikuje się z serwerem.
  - W przykładach do wykładu użyjemy bazy *PostgreSQL*.
- Najpierw przykład komunikacji z bazą bezpośrednio z Free Pascala (moduł `sqldb`).
- Potem przykład z GUI Lazarusa, z wyświetlaniem kwerendy jako tabeli `TDBGrid`.

- Free Pascal (i Delphi) posiada też w ramach tego samego API „mikro” bazy istniejące tylko w pamięci programu, z możliwością wczytania z i zapisania do pliku. `TMemDataset`, `TBufDataset`, `TSdfDataset`, `TFixedDataset` (w Delphi trochę inny `ClientDataset`).
- Na koniec, `TStringGrid` pozwala wyświetlać dane tabelarycznie bez potrzeby konstrukcji bazy danych.
- Celem wykładu jest tylko oswojenie się z koncepcją baz danych i przekonanie do sprawdzonych rozwiązań (a nie nauczenie najlepszych praktyk).

Tworzenie bazy w *PostgreSQL*

Instalacja.

- Pod Debianem:

```
$ sudo apt-get install postgresql postgresql-client
$ sudo -u postgres psql
postgres=# ALTER ROLE postgres WITH ENCRYPTED PASSWORD 'paswd';
postgres=# CREATE DATABASE my_database WITH ENCODING 'UTF-8';
postgres=# \q
```

  Ja potrzebowałem jeszcze dodatkowo (coś źle zainstalowałem?):

```
$ sudo ln -s /usr/lib/libpq.so.5.4 /usr/lib/libpq.so
```

- Pod Windowsem: pobieramy instalkę, uruchamiamy i klikamy „Next". Potem uruchamiamy *SQL Shell (psql)* wprowadzając domyślne dane logowania (klikając ENTER) i hasło jak przy instalacji. W Shellu:

```
postgres=# CREATE DATABASE my_database WITH ENCODING 'UTF-8';
postgres=# \q
```

Tabele stworzymy już łącząc się z Pascala. wiki.freepascal.org/SqlDBHowto

```
program ConnectDB;
uses sqldb, pqconnection;
var AConnection : TSQLConnection;
Procedure CreateConnection;
begin
  AConnection := TPQConnection.Create(nil);
  AConnection.Hostname := 'localhost';
  AConnection.DatabaseName := 'my_database';
  AConnection.UserName := 'postgres';
  AConnection.Password := 'paswd';
end;
begin
  CreateConnection;
  AConnection.Open;
  if Aconnection.Connected then
    writeln('Succesful connect!');
  AConnection.Close;
  AConnection.Free;
end.
```

Jeśli połączenie nie powiedzie się, dostaniemy wyjątek `EDatabaseError`.

- Komunikacja z bazą w sqldb jest oparta o klasę obsługi transakcji TSQLTransaction nawet jeśli TSQLConnection nie implementuje tak naprawdę transakcji dla danej bazy.
- Instrukcje wysyłamy przez AConnection.ExecuteDirect, kwerendy przez komponent TSQLQuery.

```
program CreateTable;
uses sqldb, pqconnection;
var
  AConnection : TSQLConnection;
  ATransaction : TSQLTransaction;
procedure CreateTransaction;
begin
  ATransaction := TSQLTransaction.Create(nil);
  ATransaction.Database := AConnection;
end;
```

```
begin
  CreateConnection;
  CreateTransaction;
  AConnection.Transaction := ATransaction;
  AConnection.Open;
  ATransaction.StartTransaction;
  AConnection.ExecuteDirect
   ('create table TBLNAMES (ID integer, NAME varchar(40));');
  AConnection.ExecuteDirect
     ('insert into TBLNAMES (ID,NAME) values (1,''Name1'');');
  AConnection.ExecuteDirect
     ('insert into TBLNAMES (ID,NAME) values (2,''Name2'');');
  ATransaction.Commit;
  AConnection.Close;
  AConnection.Free;
  ATransaction.Free;
end.
```

Z tej ↓ transakcji `ExecuteDirect` ma korzystać.

Podwójny '' oznacza ' wewnątrz tekstu.

Komunikacja z bazą w *PostgreSQL*

- Komunikacja przez `TSQLQuery`, podklasę `TDataset`.
  - Dokładniej: `TSQLQuery`→`TCustomSQLQuery`→`TCustomBufDataset`→`TDBDataset`→`TDataset`
- Kwerendę zapisujemy we właściwości `SQL` typu `TStringList`.
- `TDataset` zawiera właściwość `Fields` (o typie `TFields`, który jest jakby `TCollection` elementów `TField`) ze wszystkimi atrybutami aktualnego rekordu i funkcję `FieldByName` pobierającą `TField` danego atrybutu.
- Pomiędzy wierszami tabeli poruszamy się przez `First`, `Next`, `Prior`, `Last`, sprawdzając początek `BOF` i koniec `EOF` tabeli.
- Właściwość `PacketRecords` decyduje ile wierszy (rekordów) naraz wczytywanych z bazy: -1 oznacza wszystkie, domyślnie 10.

```
program ShowData;
uses sqldb, pqconnection;
var
  AConnection : TSQLConnection;
  ATransaction : TSQLTransaction;
function GetQuery : TSQLQuery;
begin
  Result := TSQLQuery.Create(nil);
  Result.Database := AConnection;
  Result.Transaction := ATransaction;
end;
```

```
var Query : TSQLQuery;
begin
  CreateConnection;
  CreateTransaction;
  Query := GetQuery;
  Query.SQL.Text := 'select * from tblNames';
  AConnection.Open;
  Query.Open;
  while not Query.Eof do begin
    Writeln('ID: ', Query.FieldByName('ID').AsInteger,
            ', Name: ', Query.FieldByName('Name').AsString);
    Query.Next;
  end;
  Query.Close;
  AConnection.Close;
  Query.Free;
  ATransaction.Free;
  AConnection.Free;
end.
```

↙ Właściwość `Text` z `TStrings`: nadpisz zawartość. (Np. funkcja `Add` dodałaby do zawartości.)

⟵ Wywołanie kwerendy z `SQL`.

Dopóki nie na końcu tabeli

przeskakujemy na kolejny wiersz.

Nie wprowadzaliśmy zmian więc nie wywołujemy `Commit`.

Pamiętaj: brakuje tu `try`..`finally`.

- Żeby modyfikować tabelę pod „wskaźnikiem" `TDataset` (którym jest `Query`), trzeba przejść do trybu edycji.
  - `Edit` oznacza modyfikację aktualnego wiersza.
  - `Insert` tworzy nowy wiersz przed aktualnym.
  - `Append` tworzy nowy wiersz na końcu tabeli.
- Wyjście z trybu edycji przez: `Post`, `Cancel` (anulowanie zmian), lub przejście do innego wiersza (wywoła `Post`).
- Samo `Post` nie wyśle zmian na serwer bo `TSQLQuery` jest buforowana (`TCustomBufDataset`), potrzebne `ApplyUpdates`.
- Ustawiając właściwość `ProviderFlags` atrybutów (`TField`) na `pfInKey` pozwala ręcznie wybrać pola do klauzuli `WHERE`.

```
program EditData;
uses SysUtils, sqldb, pqconnection, DB;
var
  AConnection : TSQLConnection;
  ATransaction : TSQLTransaction;
  Query : TSQLQuery;
```

```
begin
  CreateConnection;
  CreateTransaction;
  AConnection.Transaction := ATransaction;
  Query := GetQuery;
  Query.SQL.Text := 'select * from tblNames';
  Query.Open;
  Query.Edit;
  Query.FieldByName('NAME').AsString := 'Edited name';
  Query.Post;
  Query.UpdateMode := upWhereAll;
  Query.ApplyUpdates;
  AConnection.Transaction.Commit;
  Query.Free;
  ATransaction.Free;
  AConnection.Free;
end.
```

Automatycznie decyduje które atrybuty będą w klauzuli `WHERE` kwerendy.

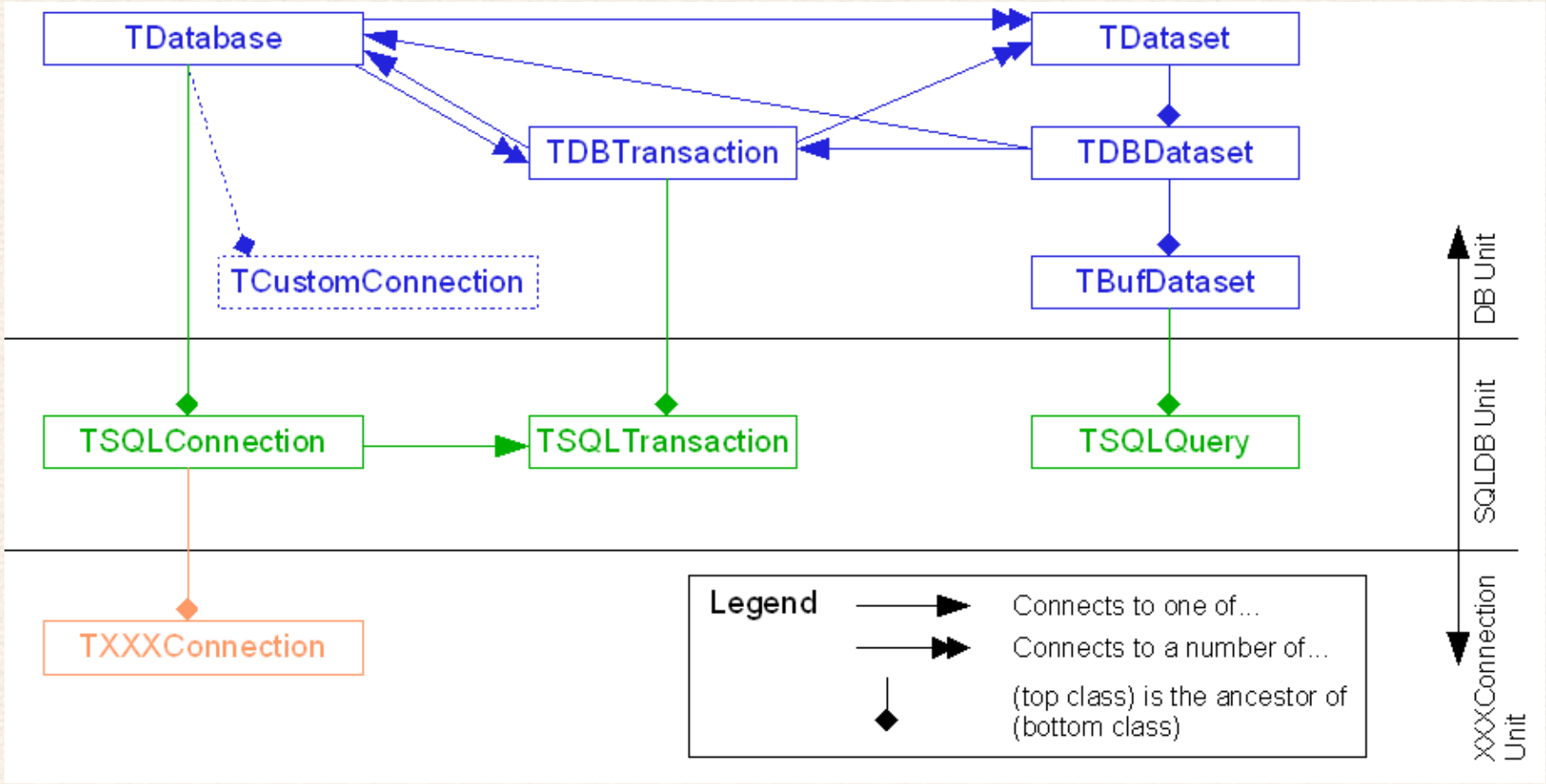
- Można użyć `Query.ExecSQL;` zamiast `Query.Open;` jeśli nie potrzebujemy danych (tzn. w roli `AConnection.ExecuteDirect`).

Kwerendy i instrukcje z parametrami

- Parametry w kwerendzie `Query.SQL` to nazwy poprzedzone `:` – podaje się konkretne wartości parametrów ustawiając `Query.Params`.
- Parametry zapewniają bardziej statyczną kontrolę typów, zmniejszają problemy z walidacją kwerendy i są szybsze (niż odbudowanie kwerendy).

```
procedure CreateTableUsingParameters;
var
  Query : TSQLQuery;
begin
  Query := GetQuery;
  Query.SQL.Text :=
    'create table TBLNAMES (ID integer, NAME varchar(40));';
  Query.ExecSQL;
  Query.SQL.Text :=
    'insert into TBLNAMES (ID,NAME) values (:ID,:NAME);';
  Query.Params.ParamByName('ID').AsInteger := 1;
  Query.Params.ParamByName('NAME').AsString := 'Name1';
  Query.ExecSQL;
  Query.Params.ParamByName('ID').AsInteger := 2;
  Query.Params.ParamByName('NAME').AsString := 'Name2';
  Query.ExecSQL;
  Query.Close;
  Query.Free;
end;
```

Podsumowanie: struktura modułów



wiki.freepascal.org/SQLdb_Programming_Reference

Obsługa baz danych w Lazarusie

- Lazarus pozwala połączyć `TDataset` z komponentem graficznym `TDBGrid` by automatycznie wyświetlać tabelę (wynik kwerendy).
- Możemy zaprogramować obiekty w kodzie albo przeciągnąć komponenty na formę i skonfigurować je w interfejsie graficznym.
  - Dodaj do projektu pakiet SQLdb, np. z menu Pakiet → Otwórz załadowany pakiet → otwórz SQLDBLaz → Use → Dodaj do projektu.
- Do obiektów które widzeliśmy wyżej dochodzi `TDBGrid` oraz, pośredniczący między gridem a `TDataset`, komponent `TDatasource`.
  - Ustaw `TDatasource.Dataset` na `SQLQuery`, oraz `TDBGrid.Datasource` na `Datasource`.

Demo:

- Zainstaluj PostgreSQL j.w.
- Stwórz bazę danych np. `my_database` j.w.
- Zaloguj się do bazy `my_database` („menu Start"→*SQL Shell (psql)* lub `sudo -u postgres psql`) i stwórz tabelę, np.

  ```
  my_database=# CREATE TABLE test_table (Name varchar(50),
  Age int, Meeting DATE, primary key (Name, Age));
  ```

  - Możesz od razu dodać wiersz do tabeli, np.

    ```
    insert into test_table (Name, Age) values ('Adam Smith',
    289);
    ```

- Otwórz Lazarus / nowy projekt, dodaj pakiet SQLdb j.w.
- Pod Windows: zapewnij, że `libpq.dll` np. z `C:\Program Files\PostgreSQL\9.1\lib` oraz `libeay32.dll`, `libintl-8.dll` i `libiconv-2.dll` z `C:\Program Files\PostgreSQL\9.1\bin` są widoczne dla programu.
  - Ja po prostu przekopiowałem je do `C:\WINDOWS\system32`.

- Dodaj do formularza z zakładki SQLdb: `TPQConnection`, `TSQLQuery`, `TSQLTransaction`.
- Dodaj do formularza z zakładki Data Access: `TDatasource`.
- Dodaj do formularza z zakładki Data Controls: `TDBGrid`.
- Ustaw właściwości `PQConnection1`: `DatabaseName: my_database`, `HostName: localhost`, `Password: paswd`, `UserName: postgres`, `Transaction` z pull-down menu (`SQLTransaction1`).
- Ustaw właściwości `SQLQuery1`: `Database` z pull-down menu (`PQConnection1`) (`Transaction` ustawi się samo), `SQL: select * from test_table;`
- Ustaw właściwości `Datasource1`: `Dataset` z pull-down menu (`SQLQuery1`).
- Ustaw właściwości `DBGrid1`: `DataSource` z pull-down menu (`Datasource1`).
- Ustaw `Connected` w `PQConnection1` na `True`, `Active` w `SQLTransaction1` na `True`, `Active` w `SQLQuery1` na `True`. Gotowe.

- Zmiany w bazie nie będą widoczne dopóki ich nie zastosujemy przez `ApplyUpdates` i `Commit`. Dodaj guzik `Apply` z następującym kodem:

```
SQLQuery1.ApplyUpdates;
SQLTransaction1.Commit;
SQLQuery1.Active:=True;
```

- Zmień właściwość `UpdateMode` w `SQLQuery1` na `upWhereAll`.
- `DBGrid` pozwala modyfikować pola i wstawiać nowe wiersze – sprawdź rezultaty w *psql*.
- Oprócz `DBGrid`, pozostałe kontrolki `DB*` pozwalają wyświetlać (i modyfikować) pola odpowiedniego typu.
  - Wstaw kontrolkę `TDBCalendar`. Ustaw w `DBCalendar1` właściwość `DataSource` na `Datasource1` i właściwość `DataField` na (w naszym przykładzie) `meeting`.

- Nie wiem dlaczego u mnie nie działa modyfikowanie daty w bazie poprzez `DBCalendar1`. Jeśli masz ten problem, wstaw `TDBEdit` (z właściwością `DataSource` ustawioną na `Datasource1` i `DataField` na `meeting` j.w.), oraz ustaw w `DBCalendar1` zdarzenie `OnChange` z kodem:

  ```
  SQLQuery1.Edit; DBEdit1.Field.Text:=DBCalendar1.Date;
  ```

  Dodatkowo ustaw właściwość `EditMask` w `DBEdit1` na `!9999-99-99` (lub odpowiednio) żeby zapobiec błędom konwersji na typ atrybutu `DATE`.

”Baza danych” w pamięci programu

- Jeśli dane potrzebujemy manipulować z i przechowywać tylko na potrzeby aplikacji, wygodniej i szybciej zamiast zewnętrznej bazy korzystać z tabel jako zwykłych struktur danych w pamięci programu.
- Klasy:
  - `TBufDataset` jest „front-endem” dla tabel z buforowaniem w pamięci,
  - `TMemDataset` z modułu `memds` jest docelowo tabelą dla baz w pamięci programu, ale obecnie jest nie zalecana, zaleca się `TBufDataset` (w przyszłości `TMemDataset` będzie dziedziczyć z `TBufDataset`),
  - `TSdfDataset` jest klasą do obsługi danych z plików CSV, w których pierwsza linia może zawierać nazwy atrybutów, a kolejne zawierają wartości atrybutów, oddzielone przecinkami lub innym ustalonym znakiem (np. tabulatorem).

C.D.N.

Kontrolki `TDBGrid`, `TStringGrid`

- Więcej szczegułów o `Grid`.

C.D.N.