Using PostgreSQL databases with Lazarus/Free Pascal.

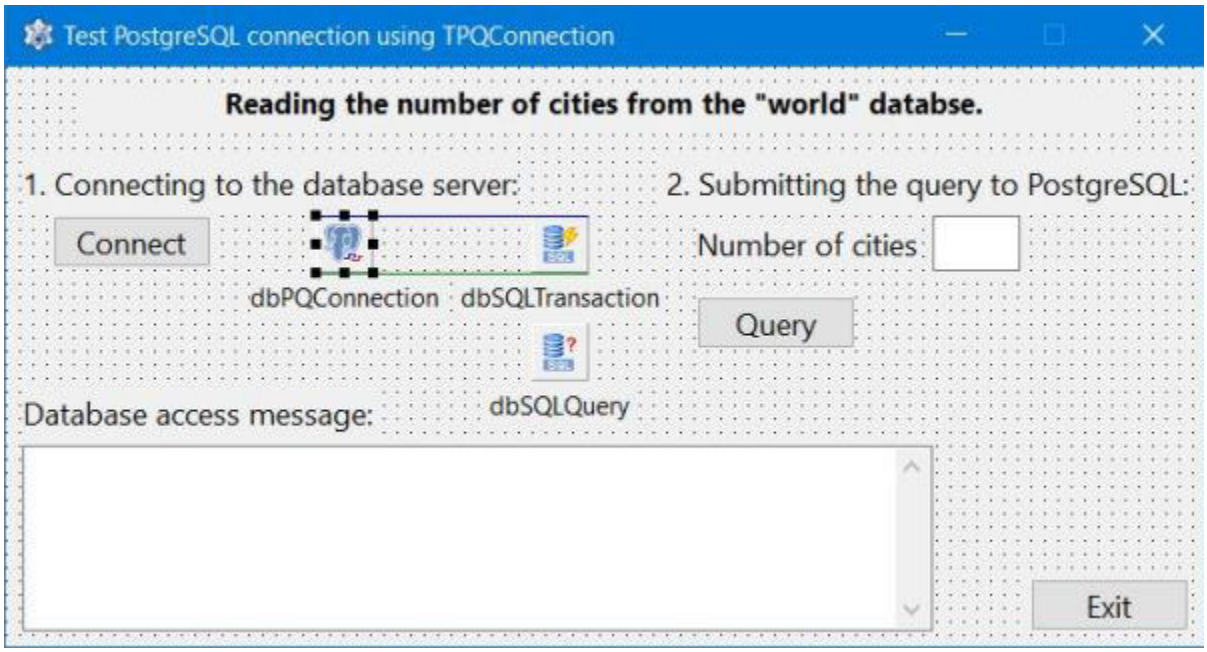
This tutorial is about the access to a **PostgreSQL 15** (64bit) database from a Lazarus/Free Pascal application; I actually use **Lazarus 2.2.6** 64bit (with FPC 3.2.2) on Windows 10. The tutorial should also apply to others versions of PostgreSQL, Lazarus and Windows.

The tutorial covers the very basics of using PostgreSQL databases from within a Lazarus/Free Pascal application, primarily how to connect to a database. In fact, working with PostgreSQL is quite the same as working with MySQL. Thus, if you need further information concerning working with relational databases in Free Pascal, my more complete tutorial Using MySQL databases with Lazarus/Free Pascal may be helpful.

The tutorial uses the **World sample database**, a port of the same-name MySQL database to PostgreSQL. If you want to reproduce the tutorial example on your computer, please, install this database and create the users as described in my Web development environment setup on MS Windows: PostgreSQL database server tutorial.

The tutorial shows, how to implement a simple Lazarus application, reading and displaying the number of cities in the "city" table of the "world" database using a **TPQConnection** object (connecting via ODBC is not covered in this text). Click the following link to download the Lazarus/Free Pascal source code of the application.

Create a Lazarus application project with the form shown on the screenshot below.

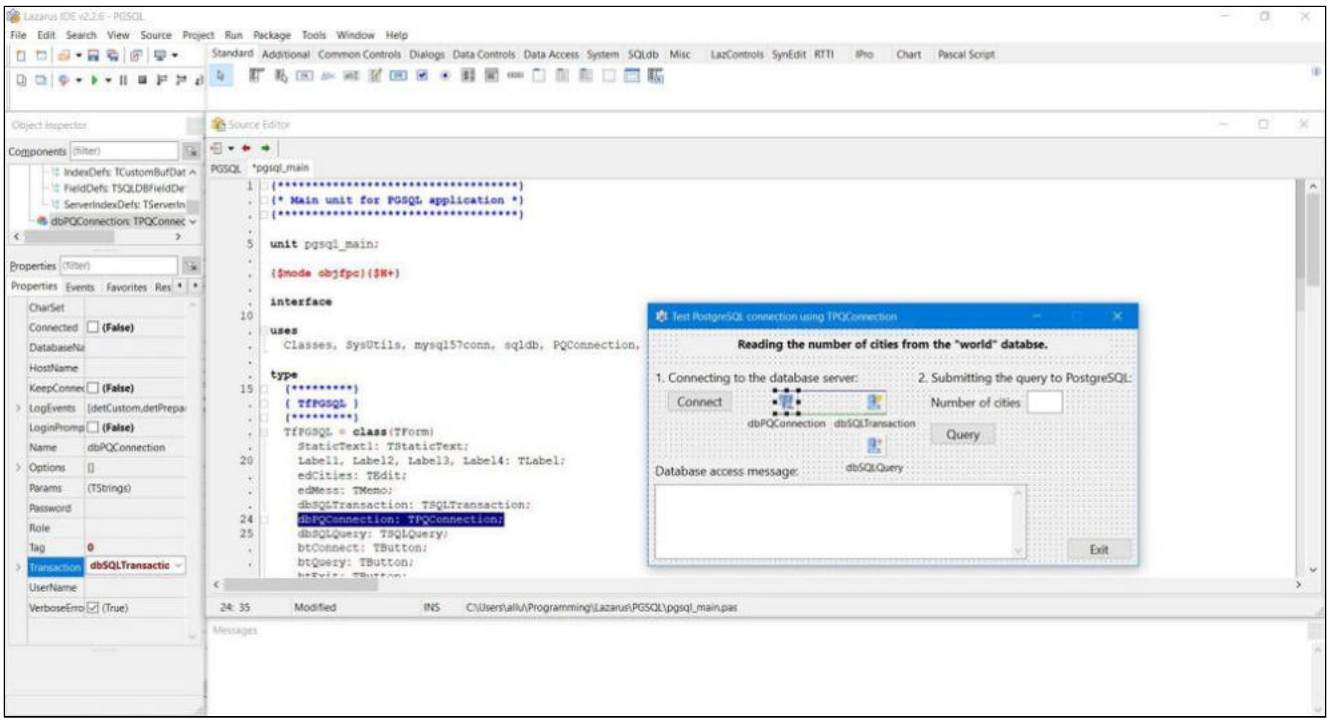


You can see on the screenshot that there are 3 database specific components (available in the Lazarus **SQLdb menu**) that have been added to the form:

- **TPQConnection** (that I named "dbPQConnection"): this is the component, that takes the requests of the TSQLQuery and TSQLTransaction components and translates them into requests specifically tailored for the PostgreSQL database.
- **TSQLTransaction** (that I named "dbSQLTransaction"): This encapsulates the transaction on the database server. A TxxxConnection object always needs at least one TSQLTransaction associated with it, so that the transaction of its queries is managed.
- **TSQLQuery** (that I named "dbSQLQuery"): This is a descendant of TDataset, and provides the data as a table from the SQL query that you submit. It can also be used to execute SQL queries (e.g. INSERT INTO, stored procedures...).

If you are familiar with using MySQL databases with Lazarus/Free Pascal, you have probably noticed that this is very similar of what you have with MySQL. In fact, the only major difference is the usage of the TPQConnection object instead of a TMySQLnnConnection object.

Concerning the **TPQConnection object**, I'll include the database specific information (host name, database name, user name and password) within the code. In the dbPQConnection property sheet, I just set "Transaction" to the name of the transaction object, in our case "dbSQLTransaction". Similarly, I'll include the SQL statements within the code, just setting the "Transaction" property of the **TSQLQuery object** in the property sheet. Doing so, its "Database" property should be automatically set to the name of the TPQConnection object's name, in our case "dbPQConnection". The "Database" property of the TSQLTransaction object should now also be set to the name of the TPQConnection object's name.



Connecting to the database.

To connect to the PostgreSQL "world" database by pushing the "Connect" button, use the following code within your TfPGSQL.btConnectClick method (fPGSQL being the name of my form, btConnect the name of my button):

```
procedure TfPGSQL.btConnectClick(Sender: TObject);
begin
  if dbPQConnection.Connected then
    dbPQConnection.Close;
  // Set the connection parameters
  dbPQConnection.HostName := 'localhost';
  dbPQConnection.UserName := 'nemo';
  dbPQConnection.Password := 'nemo';
  dbPQConnection.DatabaseName := 'world';
  // Connect to the "world" database
  edMess.Lines.Clear;
  try
    dbPQConnection.Open;
    edMess.Lines.AddText('Connection to PostgreSQL database "world" = OK!');
  except
    on E: Exception do
      edMess.Lines.AddText(E.Message);
  end;
end;
```

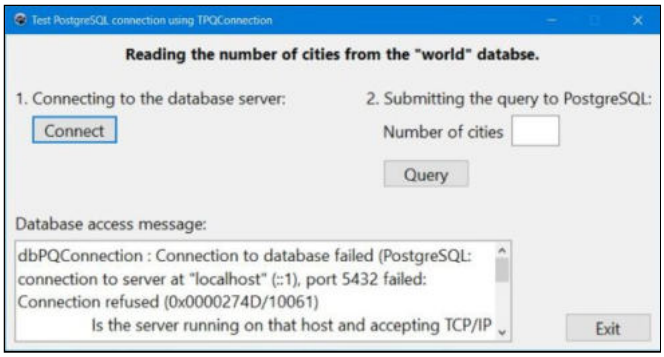
When you run the application and try to connect, you may get a similar error condition as with MySQL. In the case of PostgreSQL, the message would tell us that PostgreSQL client library *libpq.dll* can't be loaded. This would mean that *libpq.dll* cannot be found. The simplest way to resolve this issue would be to copy the DLL to the Lazarus output directory, i.e, the directory that contains the Free Pascal executable. Maybe that you wonder why I use the conditional in the last sentences. Because I actually did not get the error message. The connection succeeded without that I had to copy *libpq.dll*. I suppose (without being sure about that) that DLLs are not only searched for in the current directory and the library search path (C:\Windows\System32 and C:\Windows\SysWOW64), but

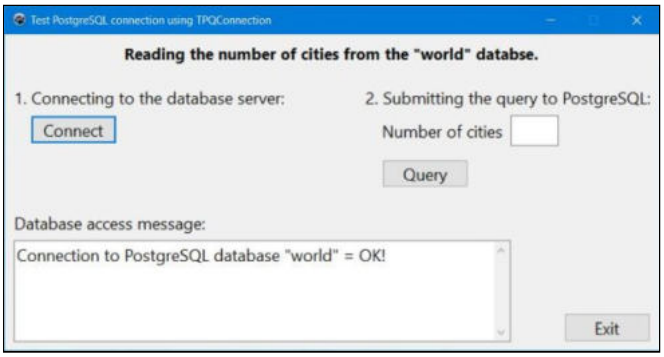
also in the executable search path (defined by the PATH environment variable), and the DLL is actually present in my C:\Program Files\PostgreSQL\15\bin, that I added to the system PATH.

Please, note that, as a difference with TMySQLnnConnection objects, **TPQConnection objects have no *Port* property**. This means that, if your PostgreSQL server listens to a port other than 5432, you'll have to use the **Params property** to set the custom port. Example (Port being the name of the variable containing the port number):

```
dbPQConnection.Params.Add('port=' + Port);
```

The screenshots below show our application after the "Connect" button has been pushed: on the left the situation where the PostgreSQL server is offline; on the right a successful connection.





Disconnecting from the database.

The disconnection from the database may be coded within the TfPGSQL.btExitClick method (activated when pushing the "Exit" button):

```
procedure TfPGSQL.btExitClick(Sender: TObject);
begin
  if dbPQConnection.Connected then
    dbPQConnection.Close;
  Close;
end;
```

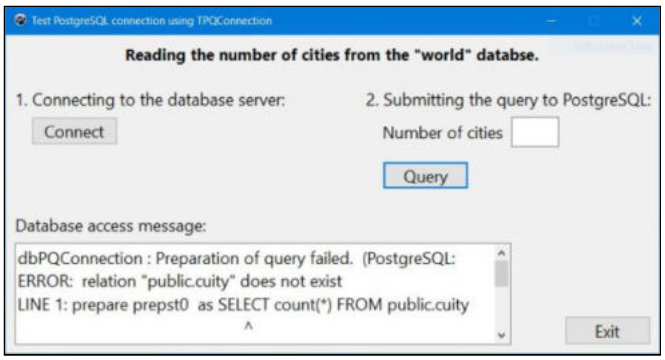
Querying the database.

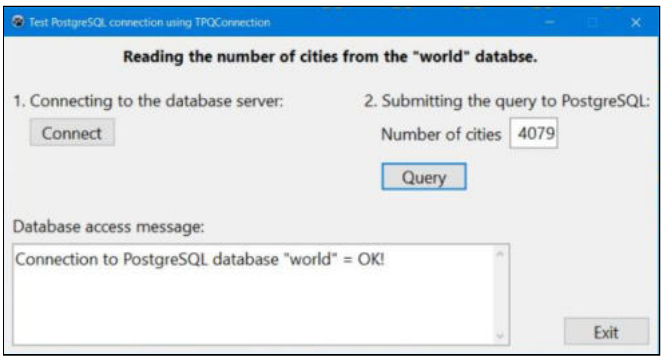
To read and display the number of records in the "city" table of the "world" database, by pushing the "Query" button, use the following code within your TfPGSQL.btQueryClick method:

```
procedure TfPGSQL.btQueryClick(Sender: TObject);
var
  Count: Integer;
begin
  if dbPQConnection.Connected then begin
    dbSQLQuery.SQL.Text := 'SELECT count(*) FROM public.cuity';
    try
      // Query the database
      dbSQLQuery.Open;
      if dbSQLQuery.EOF then
        Count := 0
      else
        Count := dbSQLQuery.Fields[0].AsInteger;
      dbSQLQuery.Close;
      // Display the query result
      edCities.Text := IntToStr(Count);
```

```
      except
        on E: Exception do begin
          edMess.Lines.Clear;
          edMess.Lines.AddText(E.Message);
        end;
      end;
    end;
  end;
```

The screenshots below show our application after the "Query" button has been pushed: on the left the situation where the table name is misspelled; on the right a successful query with the display of the number of cities.



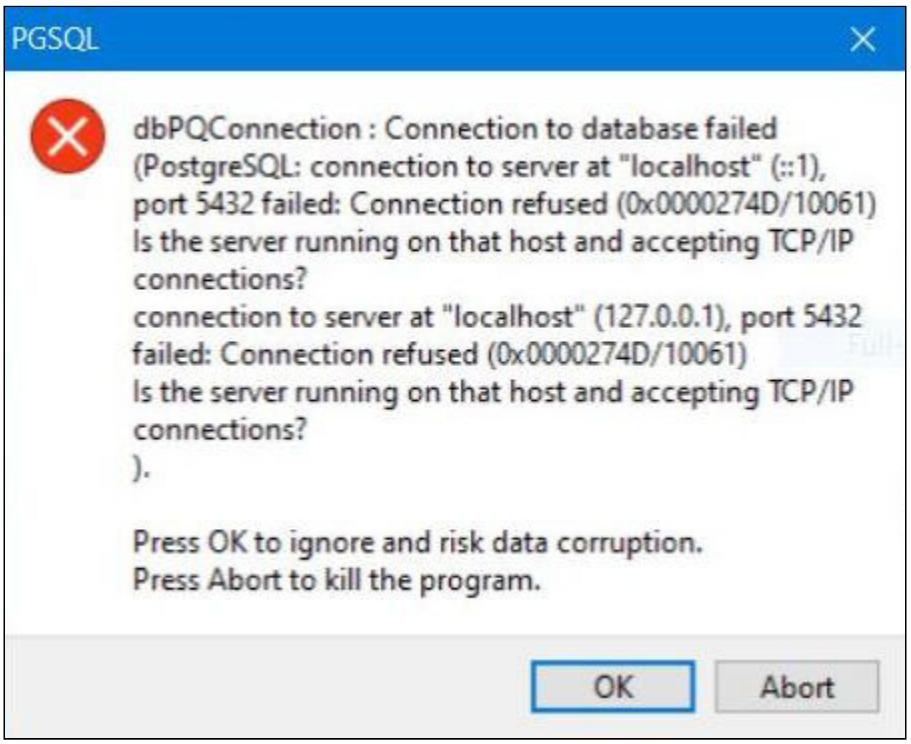


Note: When working with text, containing non-ANSI characters and you get "garbage" instead of characters with accents, try specifying the **encoding** used by the client by setting the **CharSet property** of the TPQConnection object.

The tutorial comes to its end, but there is still one difference with a TMySQLnnConnection that I want to point out. The most obvious way to code the except part of the try statement would be the following (as is with MySQL):

```
try
  ...
except
  on E: ESQLDatabaseError do
    ...
end;
```

However, this does not work! Instead of getting an error message in the corresponding field on the application form, we'd get a runtime error message, as shown on the screenshot below (case where the server is offline).



Why this? Simply because **for a TPQConnection object, ESQLDatabaseError is not fired** and if we use it in the on part of the try statement, the except part is never executed (and the message on the form will never be displayed). When an issue occurs (such as the server being offline), Windows intercepts an error that has not been treated by the application, and reacts by popping up a runtime error message box.

The workaround, that I use here, is perhaps not the best way to do (?), but it works fine. Using the structure

```
try
   ...
except
   on E: Exception do
      ...
end;
```

we catch any exception that happens during the database connection resp. the database query, we can retrieve the error message raised by PostgreSQL and display it on the form. And the application can continue in its normal way...

If you find this text helpful, please, support me and this website by signing my guestbook.